Clube Delphi

Sincronização de dados com DataSnap
Sincronize dados entre cliente e servidor com JSON

Desenvolvimento web com Intraweb
Construa uma aplicação completa

Edição 171 :: R$ 14,90

DEVMEDIA

NOVIDADES DO DELPHI BERLIN 10.1

Saiba o que há de novo na edição de aniversário

ISSN 1517990-7

MVP

R$ 1.000.000,00
INVESTIDOS EM CONTEÚDO NOS ÚLTIMOS 12 MESES.

APLIQUE ESSE INVESTIMENTO NA SUA CARREIRA...

E MOSTRE AO MERCADO QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's consumido + de **500.000** vezes

POR APENAS
R$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

DEVMEDIA

Clube Delphi

171ª Edição 2017 ISSN 1517990-7 Impresso no Brasil

Editor Geral

Joel Rodrigues (joelrlneto@gmail.com)

Jornalista Responsável

Kaline Dolabella - JP24185

Revisora Técnica

Anny Caroline (annycarolinegnr@gmail.com)

Capa e Diagramação

Romulo Araujo

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DEVMEDIA

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Joel Rodrigues - Editor da Revista
joelrlneto@gmail.com

Joel Rodrigues - **Editor Geral**

Editor da ClubeDelphi Magazine

Sumário

Artigo no estilo Solução Completa

04 – DataSnap: sincronizando dados entre cliente e servidor
[*Jones Granatyr*]

Conteúdo sobre Novidades

13 – Novidades do Delphi Berlin 10.1 Update 2
[*Fabrício Hissao Kawata*]

Artigo no estilo Solução Completa

23 – IntraWeb: Como criar uma aplicação web passo a passo
[*Gutierry Antonio*]

DataSnap: sincronizando dados entre cliente e servidor

Aprenda a sincronizar os dados entre cliente e servidor com JSON

Com o grande crescimento das aplicações mobile, uma dúvida muito comum dos desenvolvedores é onde armazenar a base de dados principal dos aplicativos. Em aplicações desktop, o comum era manter essas bases em servidores e utilizar o modelo cliente/servidor de duas camadas. Nesse modelo, a aplicação cliente acessa diretamente a base de dados, utilizando, geralmente, os componentes dbExpress ou FireDAC. Porém, em aplicativos mobile esse modelo não é muito adequado para grandes aplicações porque podem ocorrer problemas de desempenho, principalmente pelo fato desses aparelhos possuírem processamento limitado.

Neste contexto, um aplicativo móvel pode ter basicamente dois tipos de armazenamento: uma base local no próprio aparelho ou então acessar um servidor de aplicação ou webservice, que disponibiliza todos os métodos de acesso aos dados. A desvantagem de utilizar somente uma base de dados local é que os dados ficarão off-line e somente no aparelho, não existindo opções para enviá-los para um servidor. Por outro lado, caso a base esteja disponibilizada somente on-line em um servidor de aplicação, sempre será necessário existir uma conexão com a internet para utilizar o aplicativo. Com isso, uma opção viável para a criação de aplicativos é utilizar uma abordagem híbrida, na qual existem duas bases de dados: uma no cliente e outra no servidor, como se fosse um processo de replicação. Dessa forma, o usuário poderá continuar trabalhando normalmente com o aplicativo salvando as alterações locais, e quando estiver conectado, transferir seus dados para o servidor. De maneira similar, o servidor pode enviar as novas atualizações para o aplicativo cliente. Assim, o aplicativo pode superar as desvantagens supracitadas, ou seja, os dados ficarão off-line e on-line ao mesmo tempo e não haverá necessidade de conexões à internet para utilizar o software.

Fique por Dentro

Quando desenvolvemos aplicativos mobile, duas formas de armazenamento podem ser utilizadas: uma base de dados local ou uma base de dados remota. Essas duas alternativas apresentam uma série de vantagens e desvantagens, que devem ser levadas em consideração no desenvolvimento de uma nova arquitetura. Deste modo, é importante o conhecimento de um modelo híbrido para armazenamento dos dados, que possa trabalhar tanto com dados off-line quanto com dados on-line. O RAD Studio, por meio da tecnologia DataSnap, permite a construção dessas arquiteturas, que podem ser utilizadas para a sincronização entre bases de dados locais e bases de dados remotas.

Baseado nisso, o objetivo do presente artigo é mostrar como construir uma arquitetura inicial para sincronização entre bases de dados utilizando o Delphi 10.1 Berlin com o MySQL e o SQLite como sistemas gerenciadores de banco de dados. Será mostrado como construir as três partes do modelo multicamadas, ou seja: a base de dados principal (MySQL), o servidor de aplicação DataSnap e a aplicação cliente local (que armazenará seus dados utilizando SQLite). De acordo com a estrutura multicamadas, o objetivo é que o cliente acesse o servidor de aplicação, e que somente esse possa ter acesso à base de dados principal. O usuário do aplicativo poderá realizar inclusões, alterações e exclusões na base de dados local, e quando for necessário, fazer o upload dos novos registros inseridos para o servidor. De forma semelhante, quando o cliente necessitar de dados atualizados que estejam no servidor, poderá fazer o download dos novos registros para serem copiados para a base local.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Criando a base de dados local

O primeiro passo é a criação de uma aplicação multidispositivo no Delphi 10.1 Berlin (*File* > *New* > *Multi-devide application – Delphi*) utilizando o tipo *Blank Application* (todos os arquivos devem ser salvos em um diretório chamado *Cliente*). A base de dados local será criada utilizando o banco de dados SQLite, que é um dos mais utilizados para o desenvolvimento de aplicações multidispositivos com armazenamento local.

Para a geração da base de dados, utilizaremos a própria interface do RAD Studio, por meio do *Data Explorer*, o qual pode ser acessado utilizando a opção de menu *View* > *Data Explorer*. Clique com o botão auxiliar do mouse na opção *SQLite database* (apresentada ao expandir o menu FireDAC) e depois em *Add New Connection*. Depois de inserir o nome da conexão, a janela *FireDAC Connection Editor* será aberta, na qual os parâmetros da nova base de dados devem ser informados, conforme apresentado na **Figura 1**.

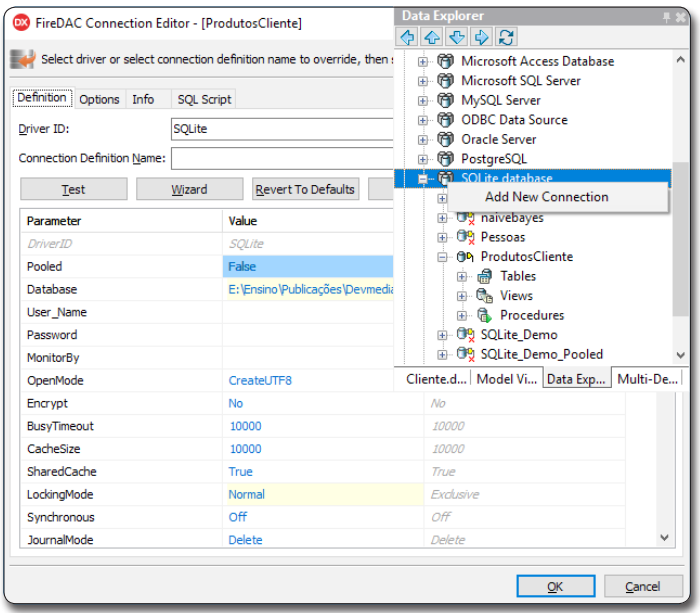

Figura 1. Criação da base de dados

Conforme mostra a figura, a propriedade *Driver ID* deve ter seu valor configurado para *SQLite*, enquanto que a propriedade *Database* deve conter o caminho onde será salvo o arquivo, bem como o nome do mesmo (*produtocliente.sdb*). Além disso, é preciso configurar o valor *Normal* para a propriedade *LockingMode* para que alterações possam ser realizadas na base. Nessa mesma janela podem ser executados os scripts SQL tanto para criação das tabelas quanto para consultas, inserções, alterações e exclusão de registros, por meio da aba *SQL Script* (veja-a na **Figura 1**). A **Listagem 1** apresenta os scripts para isso, os quais devem ser digitados nessa janela.

Entre as linhas 01 e 05 é definida a tabela de produtos, que possui uma chave primária auto incremental, e mais dois campos: um para armazenar o código do produto e outro para o nome. Por outro lado, entre as linhas 07 e 11 encontram-se os comandos para inserir cinco produtos na base de dados.

Listagem 1. Scripts para criação da tabela e inserção de dados no cliente.

```
01 create table produtos (
02   idproduto integer not null primary key autoincrement,
03   codigo char(8) not null,
04   nome varchar(20) not null
05 );
06
07 insert into produtos (codigo, nome) values ('AB123', 'Amendoim');
08 insert into produtos (codigo, nome) values ('AD754', 'Castanha');
09 insert into produtos (codigo, nome) values ('RF543', 'Leite de soja');
10 insert into produtos (codigo, nome) values ('GB237', 'Leite de coco');
11 insert into produtos (codigo, nome) values ('BB894', 'Bolacha');
```

O próximo passo é criar um novo Data Module (*File* > *New* > *Other* > *Delphi Files* > *Data Module*), chamado *dmAcessoDados*, que armazenará todos os componentes que gerenciarão a conexão do aplicativo com a base de dados recém-criada e também farão a inserção dos registros provenientes do servidor. A **Figura 2** apresenta os seis componentes necessários: *TFDConnection* (*cnnConexao*), *TFDPhysSQLiteDriverLink* (*driver*), *TFDStanStorageJSONLink* (*json*), dois *TFDQuery* (*qryProdutos* e *qryInsercao*) e um *TFDMemTable* (*memInsercao*).

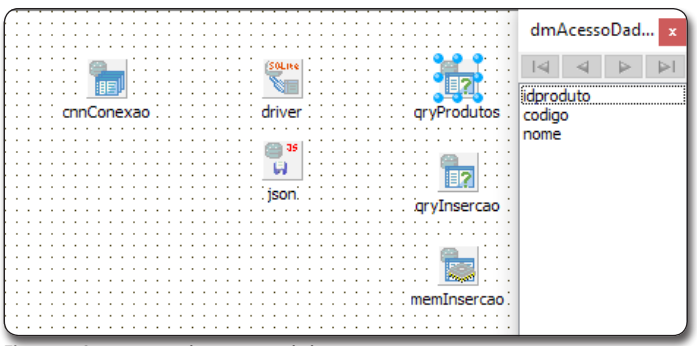

Figura 2. Componentes de acesso aos dados

O primeiro componente é responsável pela conexão do aplicativo com a base de dados e deve ter suas propriedades configuradas de acordo com a **Figura 1** (clique duas vezes no componente para abrir a janela *FireDAC Connection Editor*). Além disso, a propriedade *Login Prompt* pode ser marcada como *False* para que confirmações de usuário e senha não sejam solicitadas a cada acesso. O segundo componente diz respeito ao driver do SQLite, enquanto que o terceiro é o responsável pelo tráfego dos arquivos JSON que virão do servidor. O *qryProdutos* é o responsável pela consulta, inserção, alteração e exclusão de registros; e o *qryInsercao* é o componente que fará a inserção dos dados do servidor na base de dados local. Para isso, será utilizada uma tabela em memória (*memInsercao*), a qual será abordada posteriormente. As duas queries devem estar ligadas ao *cnnConexao* por meio da propriedade *Connection*, e a query de produtos (*qryProdutos*) deve possuir o seguinte comando em *SQL*:

```
select idproduto, codigo, nome from produtos
```

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Além disso, todos os campos devem ser adicionados no *Fields Editor*. Para isso, basta clicar com o botão auxiliar do mouse no componente e escolher a opção de mesmo nome, para depois adicioná-los por meio da opção *Add all fields*. Esses campos serão utilizados posteriormente para realizar a vinculação via Live Bindings com os componentes visuais (a **Figura 2** mostra-os ao lado da *qryProdutos*). Um último passo a ser configurado no Data Module é acessar seu evento *OnCreate* e executar a consulta SQL, utilizando o comando *qryProdutos.Open();* para que todos os dados sejam carregados quando o aplicativo for inicializado.

Criando a interface gráfica cliente

A **Figura 3** apresenta a interface gráfica do aplicativo, que é composta por duas guias: a primeira para listagem dos registros e a segunda para operações de inclusão, alteração e exclusão. Para obter o mesmo efeito da figura, deve-se adicionar ao formulário um *TTabControl* (*tabPrincipal*) com duas guias, *tabListagem* e *tabCadastro* (botão auxiliar do mouse em *tabPrincipal* > *Add TTabItem*), e com a propriedade *Align* configurada para *Client*. Na primeira guia encontra-se uma *TToolBar* (*Align* > *Top*) composta por três *TSpeedButton*: *btnDownload*, *btnUpload* e *btnIncluir*. O primeiro terá a função de fazer o download dos dados do servidor e copiar os novos registros para a base de dados local; o segundo realizará o processo inverso, ou seja, enviará os dados da base local para que os novos registros sejam adicionados à base de dados do servidor. Por fim, o último botão colocará a query em modo de inserção para que novos registros possam ser incluídos na base local.

Para obter o mesmo alinhamento da figura, o *btnDownload* e *btnUpload* devem ter sua propriedade *Align* configurada para *Left*, enquanto que o *btnIncluir* deve ter essa mesma propriedade setada para *Right*. Com relação aos ícones, os botões devem ter suas propriedades *StyleLookup* configuradas para *arrowdowntoolbutton*, *arrowuptoolbutton* e *addtoolbutton*, respectivamente. Além disso, um *TListView* (*lsvProdutos*) deve-se fazer presente abaixo da barra de ferramentas, o qual deve ser configurado com a propriedade *Align* > *Client*, para que ocupe o restante da janela, e com a *ItemAppearance* > *ItemAppearance* > *ListItemRightDetail*.

A guia cadastro (centro da **Figura 3**) também possui uma barra de ferramentas (*tlbCadastro*) com três botões: *btnVoltar*, *btnExcluir* e *btnSalvar*. Como o próprio nome sugere, o primeiro será utilizado para voltar para a guia de listagem, o segundo, para excluir um registro e o terceiro, para salvar as alterações na base de dados local. Para obter o mesmo efeito visual da figura, a propriedade *StyleLookup* desses botões deve ser configurada para *backtoolbutton*, *deleteitembutton* e *playtoolbutton*, respectivamente. Abaixo da barra de ferramentas devem estar contidos também dois *TLabel* (*lblCodigo* e *lblNome*) e dois *TEdit* (*edtCodigo* e *edtNome*), todos com a propriedade *Align* configurada para *Top*. À direita da **Figura 3** é possível observar a janela *Structure*, que mostra a hierarquia de todos os componentes que foram adicionados.

Após a adição dos componentes, faremos a vinculação via Live Bindings entre a query e o ListView e entre a query e os componentes de texto. A **Figura 4** apresenta essa vinculação, na qual podemos observar que além dos campos *codigo* e *nome*, também é necessário vincular o asterisco (*) com a propriedade *Synch* para que, quando um registro for clicado no ListView, ele seja replicado para os edits. Quando a vinculação for concluída, automaticamente serão adicionados no formulário os componentes *TBindSourceDB* e *TBindingsList*.

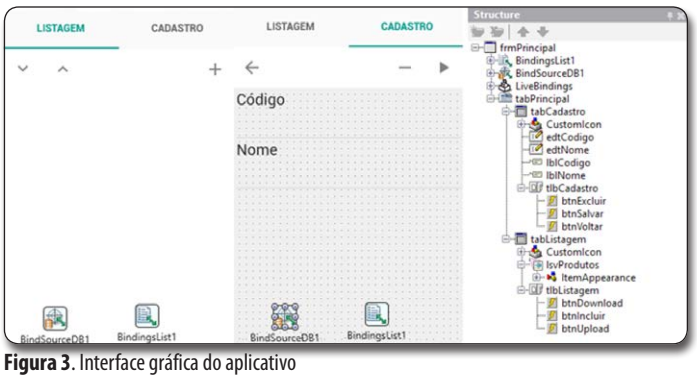

Figura 3. Interface gráfica do aplicativo

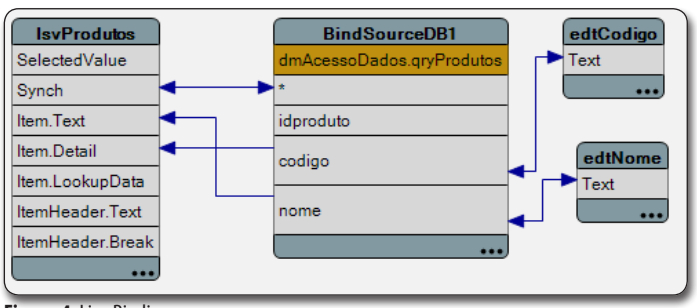

Figura 4. Live Bindings

Nota

Para que seja possível realizar o Live Bindigs, é necessário que a query esteja disponível no formulário. Por isso, deve-se utilizar a opção de menu File > Use Unit e clicar no nome do Data Module (dmAcessoDados) para que ele seja adicionado na seção uses do formulário.

Quando o usuário estiver utilizando a parte de listagem da janela, além de efetuar o download e o upload de dados, também poderá incluir novos registros ou clicar em um já existente para editar. Com isso, entre as linhas 01 e 05 da **Listagem 2** é apresentada a codificação que deve ser realizada no evento *ItemClick* do ListView, que será disparado toda vez que um registro for clicado. Na linha 03 a query é colocada em modo de edição, enquanto que na linha 04 o fluxo do aplicativo é direcionado para a guia de cadastro para que o usuário possa realizar as alterações. Em suma, quando o usuário clicar em um item o código e o nome serão exibidos nos *TEdits* da segunda guia.

De forma bastante similar, entre as linhas 07 e 11 encontra-se o código do evento *OnClick* do botão de inclusão (*btnIncluir*), que coloca a query em modo de inserção (linha 09) e faz a mudança de aba assim como no passo anterior. A diferença é que na inclusão os edits estarão em branco para que o usuário possa informar os novos dados.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 2. Edição e inserção de registros.

```
01 procedure TfrmPrincipal.lsvProdutosItemClick(const Sender: TObject;
     const AItem: TListViewItem);
02 begin
03   dmAcessoDados.qryProdutos.Edit;
04   tabPrincipal.ActiveTab := tabCadastro;
05 end;
06
07 procedure TfrmPrincipal.btnIncluirClick(Sender: TObject);
08 begin
09   dmAcessoDados.qryProdutos.Append;
10   tabPrincipal.ActiveTab := tabCadastro;
11 end;
```

O código da **Listagem 3**, entre as linhas 01 e 06, apresenta a codificação do botão voltar da segunda guia. Primeiramente, é verificado se a query está em modo de edição (linha 03) para que, caso positivo, todas as alterações sejam canceladas (linha 04). Esse processo é utilizado para que sempre que o usuário deseje gravar um registro, faça isso clicando no botão salvar. Por fim, na linha 05 o fluxo é direcionado para a primeira guia. Entre as linhas 08 e 12 está o código que deve ser programado no evento *OnClick* do botão excluir, sendo responsável pela exclusão do registro visível em tela por meio do comando *Delete* (linha 10). Similarmente ao código anterior, após a exclusão ter sido realizada, o fluxo do aplicativo volta para a primeira janela.

Nota

Em um aplicativo comercial seria necessária uma codificação adicional para exibir uma janela de confirmação do cancelamento dos dados ou exclusão de registros.

Listagem 3. Voltar para a tela anterior e exclusão de registros.

```
01 procedure TfrmPrincipal.btnVoltarClick(Sender: TObject);
02 begin
03   if dmAcessoDados.qryProdutos.State in dsEditModes then
04     dmAcessoDados.qryProdutos.Cancel;
05   tabPrincipal.ActiveTab := tabListagem;
06 end;
07
08 procedure TfrmPrincipal.btnExcluirClick(Sender: TObject);
09 begin
10   dmAcessoDados.qryProdutos.Delete;
11   tabPrincipal.ActiveTab := tabListagem;
12 end;
```

Por fim, a **Listagem 4** apresenta o código para efetuar a gravação da inclusão ou alteração dos registros na base de dados local. Na linha 03 é verificado se o campo código está vazio; caso afirmativo o usuário recebe uma mensagem em tela (linha 05) e o procedimento é abortado por meio do método *Exit* (linha 06). Entre as linhas 08 e 12 o mesmo processo é realizado, porém validando o campo nome. Nessas duas validações, a função *Trim* é utilizada para retirar espaços em branco do início e do final dos textos, com o objetivo de evitar que o usuário preencha um campo somente com espaços em branco.

Na linha 13 é verificado se a query está em modo de edição, na linha 14 as alterações são gravadas por meio do comando *Post* e, por fim, a linha 15 direciona o fluxo de execução do aplicativo para a guia de listagem.

Listagem 4. Gravação da inclusão ou alteração do registro.

```
01 procedure TfrmPrincipal.btnSalvarClick(Sender: TObject);
02 begin
03   if Trim(edtCodigo.Text) = '' then
04   begin
05     ShowMessage('O código deve ser preenchido');
06     Exit;
07   end;
08   if Trim(edtNome.Text) = '' then
09   begin
10     ShowMessage('O nome deve ser preenchido');
11     Exit;
12   end;
13   if dmAcessoDados.qryProdutos.State in dsEditModes then
14     dmAcessoDados.qryProdutos.Post;
15   tabPrincipal.ActiveTab := tabListagem;
16 end;
```

A partir desse momento, o aplicativo já está funcional e com um cadastro completo para realizar inclusões, alterações e exclusões na base de dados local. Um último detalhe a ser considerado com relação à interface gráfica é que as duas guias estão visíveis para o usuário ao mesmo tempo. Em aplicativos móveis, o ideal é que a tela exiba somente o necessário devido às limitações de espaço que os aparelhos possuem, ou seja, somente deve estar visível a guia que o usuário estiver utilizando no momento. Para chegar nesse objetivo, a propriedade *TabPosition* do *TabControl* (*tabPrincipal*) deve ser configurada para *None*. Uma programação adicional é acessar o evento *OnActivate* do formulário e digitar o código a seguir para setar a *tabListagem* como inicial:

```
tabPrincipal.ActiveTab := tabListagem;
```

Criando a base de dados no servidor

Após a aplicação cliente com a base de dados local estar concluída, iremos agora criar a base de dados em um servidor MySQL. O objetivo dessa base é também armazenar os dados de produtos, de modo que tanto o cliente quanto o servidor apresentem os mesmos dados.

A **Listagem 5** apresenta os comandos em SQL para a criação dessa base de dados, bem como a inserção dos registros. Entre as linhas 01 e 09 é criada a base juntamente com a tabela de produtos, sendo possível observar que a estrutura dessa tabela é um pouco diferente da apresentada na **Listagem 1**. O código do produto é representado pelo campo *cod*, o nome, pelo campo *descricao* e há ainda o campo *valor*, que não estava presente na tabela anterior. A estrutura dessa tabela é propositalmente diferente da anterior com o objetivo de demonstrar que as tabelas nem sempre precisam ter exatamente os mesmos campos e/ou nomes, o que pode ocorrer com frequência quando não há a necessidade de utilizar todos os atributos do servidor no cliente.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 5. Scripts para criação da tabela e inserção de dados no servidor.

```
01 create database produtosservidor;
02 use produtosservidor;
03
04 create table produtos (
05  idproduto integer not null primary key auto_increment,
06  cod char(8) not null,
07  descricao varchar(20) not null,
08  valor float
09 );
10
11 insert into produtos (cod, descricao, valor) values ('AB123','Amendoim', 6);
12 insert into produtos (cod, descricao, valor) values ('AD754','Castanha', 25);
13 insert into produtos (cod, descricao, valor) values ('AA234','Granola', 14);
14 insert into produtos (cod, descricao, valor) values ('AD754','Cerveja', 5);
```

Entre as linhas 11 e 14 estão os scripts para inserção dos dados. Note que os dois primeiros produtos (Amendoim e Castanha) encontram-se no cliente, enquanto que os dois últimos não. Deste modo, o objetivo final da sincronização é que os produtos "Leite de soja", "Leite de coco" e "Bolacha" (que estão no cliente) sejam inseridos no servidor; e que a "Granola" e a "Cerveja" (que estão no servidor) sejam incluídos na base local.

Outro ponto importante a ser observado é que ambas as bases possuem uma chave primária do tipo auto incremental, o que indica que cada uma delas gerará seus próprios valores e uma não terá influência sobre a outra. Para exemplificar, o produto "Leite de soja" pode ter seu identificador igual a 10 no servidor e igual a 5 no cliente, ou seja, esse campo não garantirá a unicidade dos registros nesta aplicação distribuída. Com isso, neste exemplo utilizaremos a combinação do código com o nome/descrição do produto para verificar se o registro já existe ou não na base de dados antes de inseri-lo, o que será feito via linhas de código.

Nota

Uma alternativa para verificar a existência dos registros é utilizar uma chave primária composta combinando o código e o nome/descrição. Neste caso, o tratamento de registros duplicados seria feito diretamente pelo banco de dados e haveria a necessidade de realizar um controle de exceções via programação para interceptar as mensagens de erro.

Criando o servidor de aplicação

O próximo passo é criar um servidor de aplicação (*File* > *New* > *Other* > *DataSnap Server* > *DataSnap WebBroker Application*) e salvar todos os arquivos em uma pasta chamada *Servidor*. Será aberto um assistente composto por cinco passos, que deve ter as seguintes opções marcadas para cada etapa: *Stand-alone application* (etapa 1), *VCL application* (etapa 2), porta 8080 (etapa 3), *ServerMethods Class* (etapa 4) e *TDSServerModule* (etapa 5). Além do arquivo do projeto (extensão *dproj*), serão também gerados três novos arquivos: *FormUnit1*, *ServerMethodsUnit1* e *WebModuleUnit1*. O primeiro apresenta uma interface gráfica para iniciar o servidor, o segundo é responsável por armazenar os métodos remotos que serão invocados pela aplicação cliente; e o terceiro representa a conexão do servidor com as aplicações clientes.

Nota

Consulte a seção **Links** para mais informações sobre as configurações de cada uma das etapas de criação de um servidor DataSnap, bem como sobre as funções de cada arquivo gerado.

A **Figura 5** apresenta os componentes que devem ser adicionados em *ServerMethodsUnit1*, sendo: um TFDConnection (*cnnConexaoServidor*), dois *TFDQuery* (*qryProdutos* e *qryInsercao*), um *TFDMemTable* (*memAuxiliar*), um *TFDStanStorageJSONLink*, um *TFDStanStorageBinLink* e um *TDFPhysMySQLDriverLink* (*driver*). Os componentes são semelhantes aos da aplicação cliente, com a diferença de que eles serão utilizados para a conexão com a base de dados MySQL (o papel das queries e da tabela em memória será explanado adiante). Ainda na **Figura 5**, à direita, é possível observar a janela *FireDAC Connection Editor*, que mostra os parâmetros *Database*, *User_Name*, *Password* e *Server* preenchidos para a conexão com o MySQL.

Nota

Caso ocorram erros de conexão com o MySQL, a propriedade VendorLib do TFDPhysMySQLDriverLink deve ser preenchida indicando o caminho da DLL cliente do banco de dados. Essa DLL chama-se libmysql.dll e pode ser encontrada na pasta de instalação do próprio MySQL.

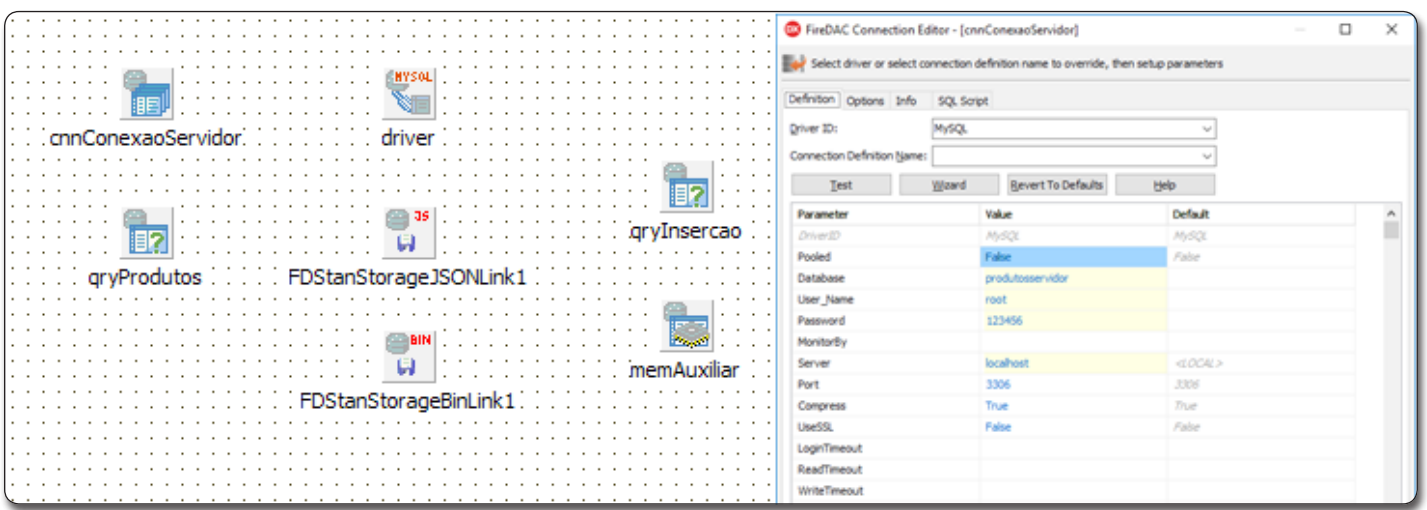

Figura 5. Componentes de acesso a dados no servidor

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Criando os métodos no servidor

O próximo passo é definir as funções que serão utilizadas para realizar o sincronismo entre as aplicações cliente e servidora. Para tanto, devem ser definidas as assinaturas dos métodos na seção *private* e *public* do arquivo *ServerMethodsUnit1* conforme mostrado na **Listagem 6**. A função da linha 02 será utilizada para verificar se um determinado produto já existe na base de dados antes de inseri-lo, evitando, assim, dados duplicados. Na linha 04 encontra-se a assinatura do método que retornará os dados do servidor para o cliente, enquanto que na linha 05 está definido o método que receberá os dados do cliente e os inserirá no servidor. Apertando-se o conjunto de teclas *CTRL + SHIFT + C*, as implementações das três funções serão criadas na seção *implementation* da unit.

Nota

Para ter acesso aos tipos de dados necessários para trafegar arquivos no formato JSON, deve-se fazer uso do namespace Data.FireDACJSONReflect, a ser inserido na seção uses.

Listagem 6. Assinatura dos métodos no servidor.

```
01 private
02  function ExisteProduto(ACodigo, ANome: String): Boolean;
03 public
04  function GetProdutos(): TFDJSONDataSets;
05  function InsereProdutosCliente(AProdutos: TFDJSONDataSets): Boolean;
06 end;
```

A **Listagem 7** apresenta o método que enviará os dados do servidor para o cliente, retornando um objeto do tipo *TFDJSONDataSets*. Na linha 03 a query é fechada para evitar pegar dados do cache, enquanto que a linha 04 apresenta o comando SQL que buscará os dados da tabela de produtos. Note que estão sendo retornados somente os campos *cod* e *descricao*, pois são esses os atributos necessários para a base de dados local. Na linha 05 é criado um novo objeto utilizando a variável de retorno *Result*, usada no comando da linha 06.

Esse comando automaticamente executa a consulta, adiciona os dados a uma variável do tipo *TFDJSONDataSets* e os empacota para transmissão pela rede, utilizando o formato JSON. Posteriormente, a aplicação cliente receberá esses dados e os inserirá na base local feita com o SQLite.

A **Listagem 8**, por sua vez, apresenta o método que verificará se um determinado produto já existe na base de dados, de modo que não sejam inseridos produtos repetidos. Conforme explanado anteriormente, serão utilizados o conjunto dos campos *cod* e *descricao* para verificar se um produto já está na base, como se fosse uma chave primária composta.

Para isso, na linha 01 a função recebe como parâmetro um código e um nome e fará um cosulta na base de dados para verificar se esse produto já existe ou não. Na linha 03 é criada uma query em tempo de execução, na linha 05 ela é associada ao *cnnConexaoServidor* e na linha 06 é definido o comando SQL que efetivamente consultará a base de dados, recebendo dois parâmetros. Nas linhas 07 e 08 os parâmetros são passados para a consulta para que na linha 09 ela seja executada.

Listagem 7. Método para enviar os produtos para o cliente.

```
01 function TServerMethods1.GetProdutos: TFDJSONDataSets;
02 begin
03  qryProdutos.Active := False;
04  qryProdutos.SQL.Add('select cod, descricao from produtos');
05  Result := TFDJSONDataSets.Create;
06  TFDJSONDataSetsWriter.ListAdd(Result, qryProdutos);
07 end;
```

Listagem 8. Método para verificar a existência de produto no servidor.

```
01 function TServerMethods1.ExisteProduto(ACodigo, ANome: String): Boolean;
02 begin
03  with TFDQuery.Create(Self) do
04  try
05   Connection := cnnConexaoServidor;
06   SQL.Text := 'select idproduto from produtos where cod = :cod
       and descricao = :descricao';
07   Params[0].AsString := ACodigo;
08   Params[1].AsString := ANome;
09   Open();
10   if IsEmpty then
11    Result := False
12   else
13    Result := True;
14  finally
15   Free;
16  end;
17 end;
```

Caso o retorno seja vazio (linha 10) será retornado o valor *False*, e caso contrário o valor retornado será *True*. Dessa forma, valores falsos indicam que o registro não existe na base de dados e que ele poderá ser inserido. Por fim, na linha 15 a query temporária é liberada da memória. É importante salientar que essa função foi declarada na seção *private* da unit, o que indica que ela poderá ser acessada somente dentro da respectiva unit. Em outras palavras, essa função não estará disponível para a aplicação cliente.

A **Listagem 9** mostra a função que fará a cópia dos dados provenientes da base de dados cliente, inserindo-os no servidor. Para isso, o método recebe como parâmetro a varíavel *AProdutos* (linha 01) do tipo *TFDJSONDataSets* e retorna um booleano que indica se obteve sucesso na cópia dos registros. Na linha 03 são definidas duas variáveis (*ACodigo* e *ADescricao*), que são utilizadas nas linhas 14 e 15 para copiar os respectivos campos da tabela em memória que possui os dados provenientes do cliente.

Na linha 05 o *Result* da função é setado para *False* — somente após todas as operações terem sido executadas sem erros é que o *Result* recebe o valor *True*, conforme a linha 24. Na linha 06, por meio da classe *TFDJSONDataSetReader*, o parâmetro *AProdutos* é lido e verificamos se ele possui algum registro, para então iniciarmos o processo de cópia dos mesmos. Para realizar essa operação foi utilizada uma tabela em memória (*TFDMemTable*), sendo que os dados em JSON são copiados para ela (linha 09) para que então, entre as linhas 12 e 23, execute-se uma estrutura de repetição

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

para realizar os *inserts* de cada registro. É importante frisar que na linha 16 é invocado o método *ExisteProduto* para executar as instruções SQL (linha 20) somente para produtos que ainda não estejam cadastrados.

Nota

O processo de cópia mostrado na **Listagem 9** executa um comando insert de cada vez, na linha 20. Uma maneira alternativa para executar essas instruções é utilizar a técnica de ArrayDML, que é mais rápida e executa todas as instruções em lote de uma só vez. Consulte a seção **Links** para mais informações sobre ArrayDML. Outra alternativa para essa sincronização é utilizar objetos da classe TFDJSONDeltas, que é bastante similar à utilização dos conceitos de DeltaDS do TClientDataSet.

Listagem 9. Método para copiar produtos do cliente para o servidor.

```
01 function TServerMethods1.InsereProdutosCliente
     (AProdutos: TFDJSONDataSets): Boolean;
02 var
03   ACodigo, ADescricao: String;
04 begin
05   Result := False;
06   if TFDJSONDataSetsReader.GetListCount(AProdutos) = 1 then
07   begin
08     memAuxiliar.Active := False;
09     memAuxiliar.AppendData(TFDJSONDataSetsReader.GetListValue
       (AProdutos, 0));
10     qryInsercao.SQL.Clear;
11     qryInsercao.SQL.Add('insert into produtos (cod, descricao)
        values (:cod, :descricao)');
12     while not memAuxiliar.Eof do
13     begin
14       ACodigo := memAuxiliar.FieldByName('codigo').AsString;
15       ADescricao := memAuxiliar.FieldByName('nome').AsString;
16       if not ExisteProduto(ACodigo, ADescricao) then
17       begin
18         qryInsercao.ParamByName('cod').AsString := ACodigo;
19         qryInsercao.ParamByName('descricao').AsString := ADescricao;
20         qryInsercao.ExecSQL;
21       end;
22       memAuxiliar.Next;
23     end;
24     Result := True;
25   end;
26 end;
```

Criando os métodos no cliente

Para que a aplicação cliente tenha acesso ao servidor, é necessário criar um cliente DataSnap no projeto, por meio do menu *File > New > Other > DataSnap Server > DataSnap REST Client Module*. Será aberto um assistente com três passos e as seguintes opções devem ser configuradas: *Local server* (etapa 1), *WebBroker stand alone server* (etapa 2) e porta 8080 (etapa 3). Serão gerados dois novos arquivos: *ClientClassesUnit1* e *ClientModuleUnit1*, os quais devem ser salvos na pasta *Cliente* juntamente com os outros arquivos do projeto. É importante enfatizar que para que esses arquivos sejam gerados o servidor precisa estar sendo executado. A **Listagem 10** apresenta a seção *private* e *public* do Data Module (*dmAcessoDados*), na qual encontram-se as assinaturas dos métodos que farão acesso ao servidor de aplicação. Apertando o conjunto de teclas *CTRL + SHIFT + C* a implementação dos três será feita na seção *implementation* da unit.

Listagem 10. Assinatura dos métodos no Data Module cliente.

```
01 private
02   function ExisteProduto(ACodigo, ANome: String): Boolean;
03 public
04   procedure AtualizaProdutosDoServidor;
05   procedure EnviaProdutosParaServidor;
06 end;
```

A **Listagem 11** mostra o método responsável por verificar se um produto já existe na base de dados local, sendo muito semelhante ao código apresentado na **Listagem 8**. A única diferença está na linha 06, na qual o comando SQL está buscando os dados da tabela de produtos no SQLite. Conforme discutido anteriormente, esses métodos tanto no cliente quanto no servidor são necessários para evitar que dados duplicados sejam inseridos nas bases de dados.

Listagem 11. Método para verificar existência de produto no cliente.

```
01 function TdmAcessoDados.ExisteProduto(ACodigo, ANome: String): Boolean;
02 begin
03   with TFDQuery.Create(Self) do
04   try
05     Connection := cnnConexao;
06     SQL.Text := 'select idproduto from produtos where codigo =
       :codigo and nome = :nome';
07     Params[0].AsString := ACodigo;
08     Params[1].AsString := ANome;
09     Open();
10     if IsEmpty then
11       Result := False
12     else
13       Result := True;
14   finally
15     Free;
16   end;
17 end;
```

A **Listagem 12** mostra o método que recebe todos os produtos que estão cadastrados no servidor, verifica se cada um deles já existe na base local e, caso não exista, executa o comando *insert*. O processo de cópia de dados é bastante similar ao que foi apresentado no servidor, com a diferença de que neste caso estamos na aplicação cliente e faremos o acesso ao servidor de aplicação para buscar os dados.

Quando criamos o cliente DataSnap, foi gerado o arquivo *ClientModuleUnit1*, o qual possui acesso a todos os métodos com a visibilidade *public* que foram definidos no servidor. Para que o acesso seja possível, deve-se adicionar em *uses* o arquivo *ClientModuleUnit1*, bem como o namespace *Data.FireDACJSONReflect*.

Na linha 03 é criada a variável *dsProdutos* do tipo *TFDJSONDataSets*, a qual recebe o retorno do método *GetProdutos* definido no servidor (linha 06). O restante do código não será explanado pois segue a mesma lógica da **Listagem 9**, ou seja, os dados são transferidos para o *FDMemTable*, um loop é feito nos registros verificando se eles já existem na base local, e, por fim, as instruções *insert* são executadas.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 12. Método para atualizar os produtos do servidor no cliente.

```
01 procedure TdmAcessoDados.AtualizaProdutosDoServidor;
02 var
03   dsProdutos: TFDJSONDataSets;
04   ACodigo, ANome: String;
05 begin
06   dsProdutos := ClientModule1.ServerMethods1Client.GetProdutos();
07   if TFDJSONDataSetsReader.GetListCount(dsProdutos) = 1 then
08   begin
09     memInsercao.Active := false;
10     memInsercao.AppendData(TFDJSONDataSetsReader.GetListValue
       (dsProdutos, 0));
11     qryInsercao.SQL.Clear;
12     qryInsercao.SQL.Add('insert into produtos (codigo, nome)
       values (:codigo, :nome)');
13     while not memInsercao.Eof do
14     begin
15       ACodigo := memInsercao.FieldByName('cod').AsString;
16       ANome := memInsercao.FieldByName('descricao').AsString;
17       if not ExisteProduto(ACodigo, ANome) then
18       begin
19         qryInsercao.ParamByName('codigo').AsString := ACodigo;
20         qryInsercao.ParamByName('nome').AsString := ANome;
21         qryInsercao.ExecSQL;
22       end;
23       memInsercao.Next;
24     end;
25   end;
26 end;
```

A **Listagem 13** mostra a implementação do método *EnviaProdutosParaServidor*, que tem a função de empacotar todos os registros da base de dados do SQLite e enviá-los para o servidor no formato JSON. Para isso, na linha 03 é declara a variável *dsProdutos* do tipo *TDFJSONDataSets*, que é utilizada na linha 06 para receber os dados da *qryProdutos*. É importante frisar que *qryProdutos* é a query utilizada em toda a janela da interface gráfica e que executa os comandos para consulta, inserção, alteração e exclusão dos registros na base de dados local. Na linha 07 é invocado o método *InsereProdutosCliente* do servidor, passando como parâmetro o *dsProdutos*, que contém todos os registros da base de dados local. Como esse método retorna um booleano, é mostrada uma mensagem para o usuário caso ocorra algum erro ao enviar os dados para o servidor, conforme a linha 08.

O último passo é programar os eventos da interface gráfica, ou seja, o botão de download e o botão de upload. A **Listagem 14** apresenta os dois eventos, sendo que entre as linhas 01 e 05 encontra-se o evento *OnClick* do botão de download enquanto que entre as linhas 07 e 10 está o código para fazer o upload dos dados, também programado no evento *OnClick*. Nas linhas 03 e 09 os procedimentos definidos no Data Module são simplesmente executados, e na linha 04 a query é atualizada para carregar os novos registros provenientes do servidor.

A **Figura 6** apresenta o resultado final da execução da sincronização, na qual pode-se observar, ao lado esquerdo, que são apresentados no ListView os produtos da base de dados local mais a "Granola" e a "Cerveja", que estavam inicialmente só no servidor. Por outro lado, à direita da figura, é possível observar a execução de uma consulta na base de dados MySQL, com os produtos "Leite de soja", "Leite de coco" e "Bolacha" copiados do cliente para o servidor.

Listagem 13. Envio dos dados para o servidor.

```
01 procedure TdmAcessoDados.EnviaProdutosParaServidor;
02 var
03   dsProdutos: TFDJSONDataSets;
04 begin
05   dsProdutos := TFDJSONDataSets.Create;
06   TFDJSONDataSetsWriter.ListAdd(dsProdutos, qryProdutos);
07   if not ClientModule1.ServerMethods1Client.InsereProdutosCliente
       (dsProdutos) then
08     ShowMessage('Erro ao enviar para o servidor');
09 end;
```

Listagem 14. Chamada dos métodos na interface gráfica.

```
01 procedure TfrmPrincipal.btnDownloadClick(Sender: TObject);
02 begin
03   dmAcessoDados.AtualizaProdutosDoServidor;
04   dmAcessoDados.qryProdutos.Refresh;
05 end;
06
07 procedure TfrmPrincipal.btnUploadClick(Sender: TObject);
08 begin
09   dmAcessoDados.EnviaProdutosParaServidor;
10 end;
```

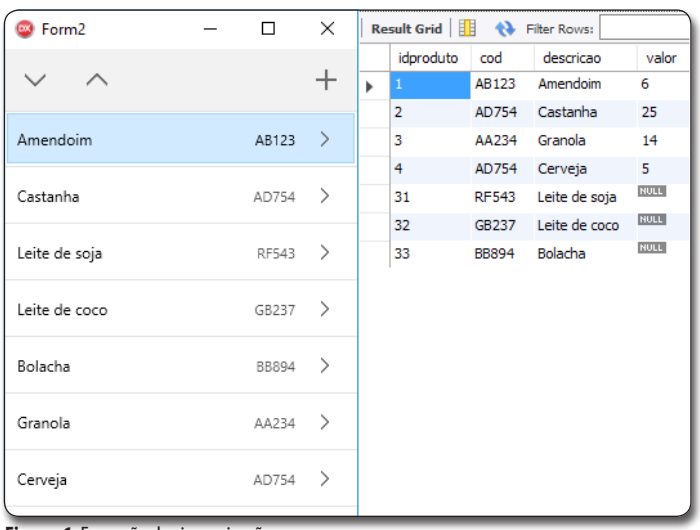

Figura 6. Execução da sincronização

Com isso, nosso aplicativo multicamadas para sincronização de dados já está funcional, ou seja, os dados tanto do cliente quanto do servidor já podem ser trafegados com as operações de sincronização. Um recurso interessante dessa estrutura é que podem existir múltiplos dispositivos clientes que estarão enviando seus dados locais para o servidor. Dessa forma, há também uma sincronização entre todos os usuários do aplicativo, visto que vários clientes podem enviar seus dados locais para o servidor, o qual inicialmente centralizará todos os registros e distribuirá para todos os usuários que estejam conectados.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Outra vantagem de uma estrutura como essa é que os usuários não precisarão estar conectados a todo momento no servidor, de modo que possam utilizar os recursos do aplicativo localmente e, quando conectados, realizar a sincronização e buscar por atualizações.

O presente artigo mostrou somente o processo de inclusão de novos registros, ficando de fora as operações de alteração e exclusão. No que diz respeito à alteração, seriam necessários novos métodos que recebessem os dados e realizassem um comando *update* nas bases de dados, muito semelhante ao que foi aqui apresentado. Já com relação à exclusão, haveria necessidade de um controle mais elaborado, pois quando um usuário exclui um registro seria preciso verificar se esse registro realmente deveria ser apagado da base de dados principal no servidor e também de todos os dispositivos dos usuários. Isso poderia levar a problemas, como a exclusão de um produto no servidor e a venda desse produto em um aplicativo cliente off-line. Todas essas questões precisam ser levadas em consideração no desenvolvimento de uma aplicação distribuída como essa.

Autor

Jones Granatyr

Doutorando em Informática bolsista CAPES e Mestre em Ciência da Computação bolsista CNPq, ambos na área de Inteligência Artificial. É fundador do portal IA Expert e trabalha em projetos de pesquisa relacionados à área de Inteligência Artificial, tais como Sistemas Especialistas, Mineração de Dados, Mineração de Textos e Sistemas Multiagente.

Links e Referências Bibliográficas:

Documentação da Embarcadero
http://docwiki.embarcadero.com

DataSnap: Copiando dados para bases locais com Array DML
http://www.devmedia.com.br/datasnap-copiando-dados-para-bases-locais-com-array-dml/37476

Como buscar dados de servidores DataSnap utilizando JSON
http://www.devmedia.com.br/como-buscar-dados-de-servidores-datasnap-utilizando-json/37361

Como depurar aplicações DataSnap
http://www.devmedia.com.br/como-depurar-aplicacoes-datasnap/37277

J. Granatyr. "Multicamadas e REST com DataSnap", Clube Delphi, v. 170, 2016.

J. Granatyr, G. Pauli, T. L. Lauxen, J. P. Barddal, F. Spak, F. Taffe. "Construindo Aplicativos Android no Delphi 10 Seattle", Clube Delphi, v. 166, 2016.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Novidades do Delphi Berlin 10.1 Update 2

Saiba o que há de novo na Anniversary Edition

Tendo em vista seu cenário mais recente, por definição, o Delphi pode ser enquadrado em duas categorias distintas, sendo considerado tanto um ambiente de desenvolvimento integrado (IDE — *Integrated Development Environment*) quanto uma efetiva linguagem de programação. Como linguagem, seu dialeto tem como base o Object Pascal, que foi a linguagem primária da ferramenta, desde o momento de sua concepção há décadas atrás.

É fato que a partir de sua versão XE5, de forma invariável, o Delphi acabou por ganhar mais uma definição, SDK (*Software Development Kit*). Essa sigla em inglês, muito popular no contexto de desenvolvimento de software, faz referência a um conjunto de recursos que contribuem diretamente no apoio à criação do software final. Logo, o suporte completo ao desenvolvimento multiplataforma provido pelo Delphi mais recentemente, o caracterizam como um verdadeiro kit de desenvolvimento. Fundamentalmente, seria um SDK provendo acesso a outros SDKs, tais como Android SDK e Windows SDK. Logo, na prática, isso significa dizer que somente a sua própria instalação já habilita a construção de aplicativos para todas as plataformas suportadas: Windows, MacOS, iOS e Android.

Delphi Berlin 10.1

Berlin é o codinome da mais recente versão estável do Delphi, a 10.1, uma evolução do Delphi 10 Seattle. Lançada já pela IDERA, empresa que adquiriu a Embarcadero (antiga detentora dos direitos do IDE), essa grande atualização marcou a adição de várias novas *features* ao produto, bem como a estabilização de outras já existentes.

Diante disso, a partir deste ponto, o pleno conhecimento das possibilidades que a ferramenta pode proporcionar ao desenvolvimento de aplicações é relevante. Basicamente, a ideia é que o desenvolvedor Delphi tenha a expertise necessária sobre o poderio do IDE em seu plano atual, facilitando o conhecimento das nuances de um novo release. Seguindo por essa linha, alguns pontos fundamentais devem ser levados em conta:

Fique por Dentro

Historicamente, a chegada de uma nova versão do Delphi implica no acréscimo de novas pontualidades a serem estudadas. Adicionalmente, para se manter atualizado com as novidades é imprescindível o conhecimento fundamental sobre o estágio atual da ferramenta até aquele momento. Dito isso, este artigo, ao mesmo tempo em que mostra as novidades trazidas pelo Update 2 do Delphi Berlin, pondera também sobre o cenário corrente do produto até este momento e o que esperar dele num futuro próximo.

Android

Atualmente, o desenvolvedor Delphi é capaz de construir aplicações para as principais versões do Android, incluindo KitKat (4.4), Lollipop (5.1) e Marshmallow (6.0), o que propicia ao profissional uma abertura de mercado considerável. Adicionalmente, ainda se faz presente a questão do reaproveitamento de conhecimento proporcionado pela ferramenta: seja para a construção de uma aplicação VCL Desktop ou uma aplicação FMX Mobile, a Delphi Language será a utilizada.

VCL

Visual Component Library, ou simplesmente VCL, é o principal framework de desenvolvimento do Delphi, tendo sua concepção atrelada ainda à época da Borland. Desde aquela época sua evolução tem se mostrado consistente, sempre caminhando paralelamente ao Windows, que é a sua plataforma de destino. Em vista disso, com o lançamento da versão 10 do sistema operacional, surge uma nova gama de componentes visuais, todos voltados a atender o novo Windows.

TRelativePanel é um desses componentes, e sua função básica é prover uma capacidade relativa de posicionamento visual dos elementos em tela, tal como ocorre em aplicações web e mobile mais modernas. Em termos práticos, ao invés de se posicionar um determinado controle de forma absoluta e fixa, o TRelativePanel possibilita, por exemplo, que se determine que o elemento A deva estar posicionado sempre à esquerda de um elemento B, ou mesmo que um elemento X deva se posicionar sempre ao topo de outro elemento Y, e assim por diante.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

A **Listagem 1** exibe um trecho de código DFM que exemplifica o uso prático do componente. Basicamente, o TRelativePanel mantém uma coleção de controles, neste caso, um botão (Button) e uma caixa de entrada (Edit), por meio de sua propriedade *ControlCollection*. O primeiro é então alinhado ao fundo e centralizado horizontalmente. A relatividade de seu posicionamento é com relação ao próprio Panel, conforme indicado pela configuração das propriedades *AlignBottomWithPanel* e *AlignHorizontalCenterWithPanel*, respectivamente.

Já o Edit tem seu posicionamento relativo relacionado tanto ao Panel quanto ao Button. Sendo assim é determinado que ele fique sempre numa posição acima do botão (propriedade *Above*) e alinhado horizontalmente ao centro do Panel (propriedade *AlignHorizontalCenterWithPanel*).

Listagem 1. Código DFM da utilização de um componente TRelativePanel

```
01 object RelativePanel1: TRelativePanel
02 ...
03 ControlCollection = <
04   item
05     Control = Button1
06     AlignBottomWithPanel = True
07     AlignHorizontalCenterWithPanel = True
08 ...
09   end
10   item
11     Control = Edit1
12     Above = Button1
13     AlignHorizontalCenterWithPanel = True
14   end>
15 ...
```

EMS

No contexto do Delphi, EMS é uma sigla que foi introduzida recentemente e remete a Enterprise Mobility Services, uma solução que expõe uma camada de serviços por meio de APIs REST. Em termos gerais, o EMS pode ser enquadrado na mesma linha de soluções do DataSnap, que, além de incluir o conceito de camadas em sua arquitetura, estabelece o atendimento de clientes por meio de um servidor centralizado, que concentra dados e regras de negócio e expõe os serviços necessários. A **Figura 1**, retirada do próprio site da ferramenta, ilustra de forma concreta a intenção do EMS mediante a apresentação de sua arquitetura.

Internet das Coisas

Internet das Coisas, Internet of Things ou simplesmente IoT é a mais nova revolução tecnológica disseminada no cenário atual. Presente no dia a dia, a IoT é a capacidade de se conectar dispositivos eletrônicos a fim de se obter alguma experiência de uso. Um exemplo clássico seria a possibilidade de se controlar as luzes ou o acionamento do portão eletrônico de uma residência, usando para isso um smartphone ou tablet.

Visando atender cenários como esse, o Delphi atualmente contempla uma série de recursos que, uma vez empregados em aplicações mobile ou desktop, proveem a integração de dispositivos, sensores, serviços na nuvem e dados corporativos. Um exemplo disso é o seu suporte a BeaconFence, uma solução de proximidade geográfica que provê um rastreamento preciso de informações de localização, sem o uso de GPS. Outro exemplo é o recurso de Tethering, que prepara as aplicações desktop e mobile para cenários que envolvam o controle remoto de aplicativos, conectividade via Bluetooth, acionamento de outros dispositivos, etc.

Assim como não poderia deixar de ser, tanto para o uso de Beacon quanto para Tethering, o Delphi dispõe de componentes pré-elaborados, devidamente disponibilizados na Tool Palette (**Figura 2**).

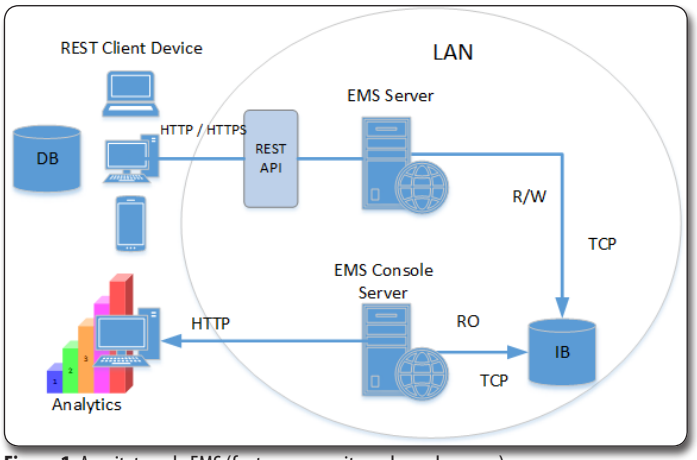

Figura 1. Arquitetura do EMS (fonte: community.embarcadero.com)

Figura 2. Componentes nativos para Beacon e Tethering

Banco de dados

Em meados de 2013 a Embarcadero anunciava a compra do AnyDAC, uma biblioteca de terceiro que contemplava não só componentes, mas também uma solução completa para acesso a dados. O resultado dessa experiência foi o surgimento do FireDAC, novo nome dado à biblioteca, uma vez que essa passou a ser parte integrante do contexto nativo do Delphi.

A partir dessa adição, a excelência no suporte aos bancos mais tradicionais, tais como Oracle, SQL Server, MySQL e Firebird, foi mantida. Todavia, a evolução do FireDAC tem sido constante e, no cenário atual do Delphi Berlin, esse suporte se expandiu a bases de dados mais modernas, como SQLite e MongoDB.

SQLite pode ser definido como um banco de dados embutido ou embarcado, que não exige um processo de gerenciamento separado. Isso o torna uma solução leve o suficiente para atender cenários mais enxutos, como é caso de aplicativos mobile. Em vista disso, sua utilização é quase que "nativa" em aplicações Android, por exemplo. Já o MongoDB entra na classificação de bancos NoSQL, definido assim por sua característica não relacional, sem esquemas e orientada a documentos.

Delphi Berlin 10.1 - Update 2 — Anniversary Edition

Essa atualização, batizada como edição de aniversário (Anniversary Edition), chega com um sabor especial uma vez que vem para complementar de vez os recursos trazidos pelo próprio Berlin.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Se fosse necessário escolher um único termo para ilustrar as novidades trazidas por essa atualização, esse termo seria "Windows 10". Em suma, isso significa dizer que você, como desenvolvedor, poderá construir uma aplicação direcionada à nova plataforma da Microsoft e de quebra publicá-la na Windows Store.

Obviamente, o Berlin Update 2 não fica restrito a isso — traz consigo outras novidades, aprimoramentos e correções de bugs do produto. Como exemplo, há a adição de novos controles VCL para calendário, novos estilos (Windows 10 styles) para VCL e FireMonkey Applications, inclusão do Quick Edit para melhoria de produtividade, bem como o suporte avançado para iOS10 e MacOS Sierra.

Instalação

Esse update apresenta algumas peculiaridades percebidas já na instalação. Um ponto inicial destacado pela própria fabricante é sobre quem está apto a instalar essa edição de aniversário. Basicamente são dois grupos de usuários: aqueles que adquirem a ferramenta no modo de avaliação (Trial) e clientes que possuem uma assinatura de atualização (Update Subscription) ativa do produto.

Já olhando para a prática, a instalação do Update 2 implica também na prévia desinstalação de qualquer atualização anterior existente, no caso, o Update 1. Sendo assim, ao executar o instalador num ambiente em que haja um Delphi Berlin instalado, uma mensagem é exibida, tal como mostra a **Figura 3**.

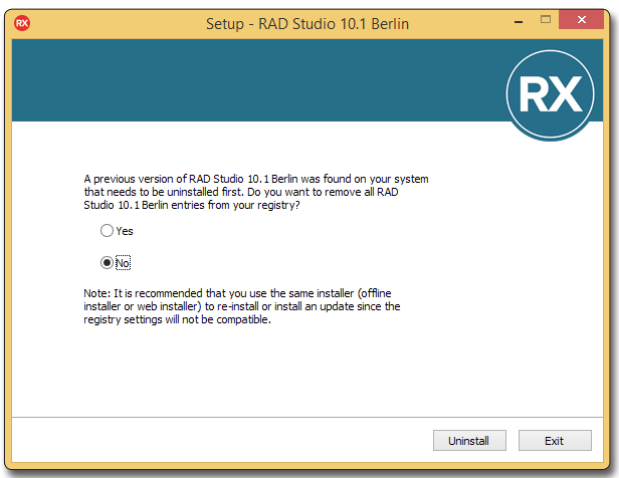

Figura 3. Instalação do Update 2 — mensagem de versão prévia encontrada no ambiente

Em complemento, ainda sobre a instalação, a **Figura 4** mostra a tela de seleção de plataformas dessa versão. Através dela, o desenvolvedor é capaz de definir exatamente a plataforma de destino que se quer alcançar usando a ferramenta: Windows 32 bits, Windows 64 bits, iOS e Android.

Por fim, tendo em vista o novo suporte à construção de aplicativos para Windows 10, o instalador do Berlin Update 2 contempla uma etapa adicional exclusiva, referente à instalação do Windows Software Development Kit (**Figura 5**). Conforme o próprio nome sugere, esse é o SDK responsável por prover a capacidade de construção para a referida plataforma.

Figura 4. Instalação do Update 2 — tela de seleção de plataformas

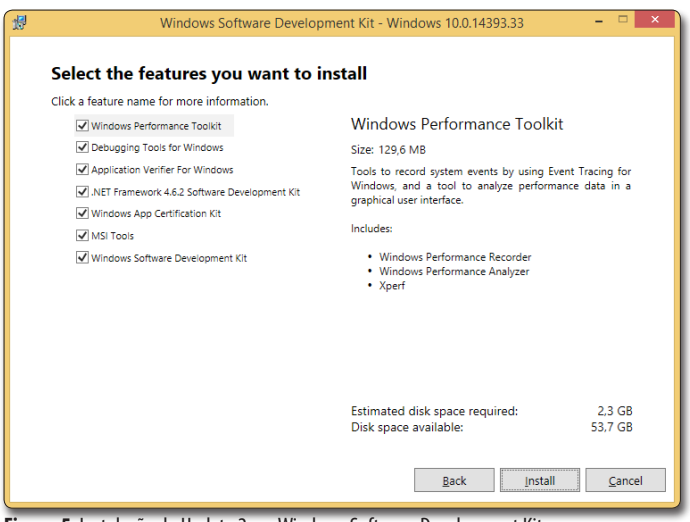

Figura 5. Instalação do Update 2 — Windows Software Development Kit

Platform Assistant

Platform Assistant é o nome do recurso que o Delphi utiliza no desenvolvimento multi-device com plataformas heterogêneas — Windows 64 bits, OS X e iOS —, provendo a depuração e execução remota das aplicações. Assim como ocorre em quase todo novo release, o Berlin Update 2 traz consigo novas versões dos instaladores de origem e de destino do Platform Assistant (*setup_paserver.exe* e *PAServer18.0.pkg*). Esses arquivos ficam disponíveis na pasta *PAServer* do diretório de instalação do IDE, tal como mostra **Figura 6**.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

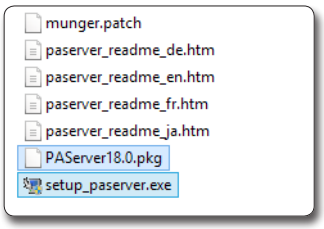

Figura 6. Novos instaladores do Platform Assistant

No contexto Apple isso se torna de extrema importância, e o uso da versão do Platform Assistant condizente com a versão corrente do IDE é um fator essencial para o sucesso nesse tipo de desenvolvimento. Neste caso, apesar de não ser mandatório, é recomendada pela IDERA/Embarcadero a desinstalação prévia das versões antigas do Platform Assistant tanto no ambiente Windows quanto no ambiente Mac e posterior instalação da versão que acompanha o Delphi Berlin Update 2.

Novos controles de calendário para VCL

Conforme mencionado inicialmente, nessa sua nova versão o Delphi acrescenta à sua já extensa gama de componentes VCL mais dois novos elementos, TCalendarView e TCalendarPicker. Ambos são concebidos para o Windows 10, uma vez que simulam os controles nativos WinRT da plataforma. Apesar de terem sido criados com essa intenção, eles oferecem suporte às versões mais antigas do sistema operacional, garantindo sua usabilidade em qualquer tipo de aplicação relacionada. Além disso, eles vêm habilitados com suporte completo ao uso de estilos (VCL Styles). No IDE, ambos ficam disponíveis numa guia exclusiva da Tool Palette, denominada Windows 10 (**Figura 7**).

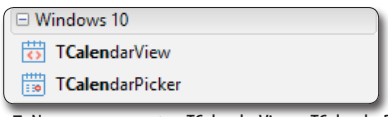

Figura 7. Novos componentes: TCalendarView e TCalendarPicker

TCalendarView

O componente TCalendarView provê três tipos de visões diferentes do calendário — por mês (Month), por ano (Year) e por década (Decade) — conforme ilustrado na **Figura 8**. A opção por uma ou outra é feita por meio de sua propriedade *DisplayMode*, que disponibiliza um enumerado de valores: *dmMonth*, *dmYear* e *dmDecade*, sendo o primeiro o valor padrão.

Durante a execução da aplicação também é possível alternar por entre essas visões, melhorando a experiência do usuário. Por exemplo, um clique no cabeçalho da visão por mês faz com que seja aberta a visão por ano. Na mesma medida, um clique no cabeçalho da visão por ano faz com que seja aberta a visão por década.

Em termos de código, o TCalendarView apresenta alguns eventos relacionados diretamente ao tratamento de dia, mês e ano, sendo eles: *OnDrawDayItem*, *OnDrawMonthItem* e *OnDrawYearItem*. Assim, por meio deles é possível, por exemplo, exibir ao usuário todos os domingos numa cor diferente dos demais dias da semana.

A **Listagem 2** faz isso através da manipulação do evento OnDrawDayItem:

Listagem 2. Manipulando o evento OnDrawDayItem para mudar a cor dos domingos

```
01 procedure TForm1.CalendarView1DrawDayItem(Sender: TObject;
02   DrawParams: TDrawViewInfoParams; CalendarViewViewInfo: TCellItemViewInfo);
03 begin
04   //verifica se é o primeiro dia da semana (domingo)
05   if DayOfWeek(CalendarViewViewInfo.Date) = 1 then
06     //muda a cor do dia para verde no controle de exibição
07     DrawParams.ForegroundColor := clGreen;
08 end;
```

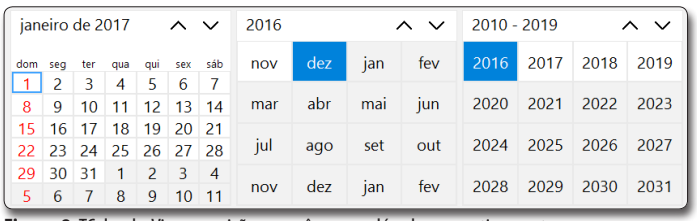

Figura 8. TCalendarView — visão por mês, ano e década, respectivamente

Além da facilidade em manipular datas, o grande apelo do TCalendarView é sua experiência visual. Assim, uma porção de opções de customização são disponibilizadas por meio de suas propriedades. A seguir são enumeradas cinco delas:

- *HeaderInfo*: permite mudar a cor e a fonte do cabeçalho do calendário, com o resultado podendo ser visualizado ainda em tempo de design;
- *FirstYear* e *LastYear*: usados para determinar os anos inicial e final do calendário. O valor padrão é 0, que indica que não há limitação;
- *MinYear* e *MaxYear*: usados para indicar os anos mínimo e máximo que o usuário poderá selecionar no calendário. O valor padrão é 0, que indica que não há limitação;
- *SelectionMode*: permite definir se o usuário poderá selecionar nenhuma (*smNone*), uma (*smSingle*) ou várias datas simultaneamente (*smMultiple*);
- *FirstDayOfWeek*: indica qual dia da semana será mostrado como o primeiro dia da semana no calendário. Essa configuração é comumente utilizada em contextos comerciais, de cultura e nacionalidade.

TCalendarPicker

O TCalendarPicker complementa a série de novidades relativas a componentes de calendário para a VCL. Em termos visuais, ele se apresenta como um controle de edição comum (**Figura 9**) e, a grosso modo, se assemelha ao clássico controle TDateTimePicker, tradicional da VCL. A diferença aqui se dá pela sua estilização visual e usabilidade. Em execução, para a seleção de uma data, o TCalendarPicker abre um pop-up muito parecido com o TCalendarView, como pode ser visto na **Figura 10**.

Assim como ocorre com o TCalendarView, grande parte de suas propriedades diz respeito à sua customização visual. Além disso, muitas delas se repetem, como *FirstYear*, *LastYear*, *MinYear*, *MaxYear* e *FirstDayOfWeek*. Ainda em termos comparativos, aqui a

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

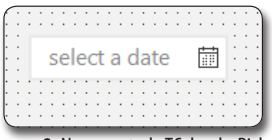

Figura 9. Novo controle TCalendarPicker

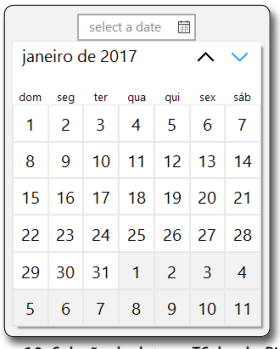

Figura 10. Seleção de data no TCalendarPicker

opção *HeaderInfo* é apresentada como *CalendarHeaderInfo* e não há a possibilidade de multisseleção de datas, tal como se a propriedade *SelectionMode* tivesse o valor fixo de *smSingle*. Além disso, como padrão, o componente apresenta seu hint em inglês ("select a date"), todavia isso pode ser facilmente configurado por meio de sua propriedade *TextHint*.

Novos Windows 10 Styles

No contexto das aplicações Delphi, um estilo ou style é uma forma de se mudar a aparência da aplicação, incluindo desde os seus detalhes gráficos até seu look-and-feel. Em vista disso, no popular, um estilo pode ser equiparado a um tema do Windows. Em suas versões mais antigas, o Delphi não contava nativamente com um recurso desse tipo, logo, na necessidade de uso, era necessário recorrer a soluções de terceiros, o que acarretava custos, dependências e erros indesejados.

Em sua versão XE2 o Delphi trouxe consigo o recurso de suporte a estilos de forma nativa, recurso esse que é denominado VCL Styles. A partir disso, a cada novo lançamento do IDE, novos estilos são incorporados à ferramenta, tornando o leque de opções visuais ainda mais extenso para o desenvolvedor. Seguindo por essa linha, no Berlin Update 2, seis novos estilos foram incorporados, sendo três para VCL e três para FireMonkey (FMX), todos eles relacionados ao Windows 10.

De uma forma geral, esses estilos acompanham o Delphi na forma de arquivos e ficam alocados no diretório padrão:

C:\Users\Public\Documents\Embarcadero\Studio\18.0\Styles

Entre a lista de arquivos, Win10ModernSlateGray, Win10ModernPurple, Win10ModernGreen, Windows10SlateGray, Windows10Purple e Windows10Green são os que se referem aos novos estilos dessa versão.

Na prática, tomando como base uma aplicação VCL tradicional, a aplicação de um estilo se dá por meio da opção *Appearance*, disponível no menu *Project* > *Option* > *Application*. Clicando nessa opção, uma lista com todos os estilos disponíveis será apresentada, assim como mostrado na **Figura 11** (note nessa figura os novos styles).

Já a **Figura 12** mostra uma aplicação em runtime estilizada com o novo Windows10 Green style.

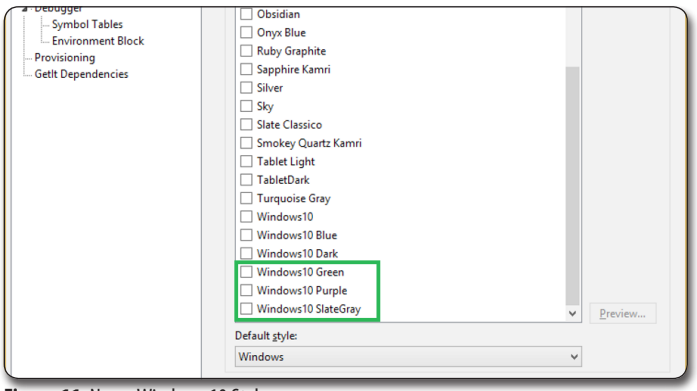

Figura 11. Novos Windows 10 Styles

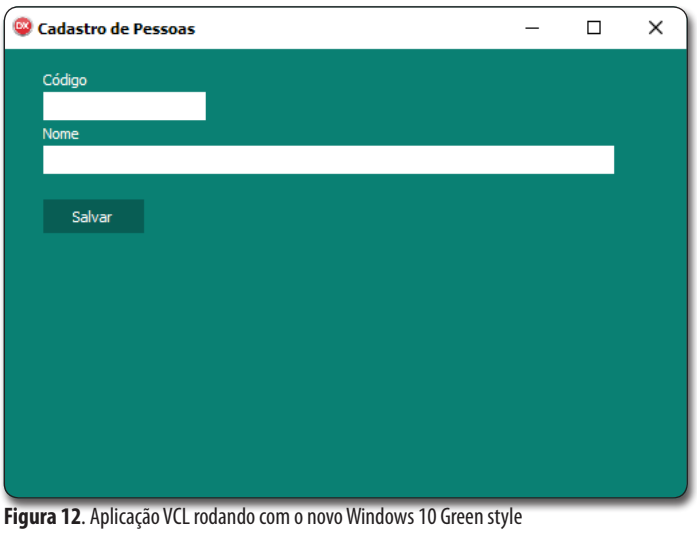

Figura 12. Aplicação VCL rodando com o novo Windows 10 Green style

Quick Edit

Quick Edit é o nome dado ao novo recurso do Delphi, introduzido no Berlin Update 2, que tem como proposta aumentar a produtividade do desenvolvimento no que diz respeito a execução de ações comuns em tempo de projeto no VCL Form Designer.

Passando isso para o cotidiano do desenvolvedor, o Quick Edit possibilita a edição rápida dos controles em tela, por meio da configuração de suas propriedades mais essenciais, tais como *Name*, *Caption*, *Alignment*, *Layout* e *Color*. Na prática, a acessibilidade ao Quick Edit de um controle é feita por meio de seu menu de contexto, conforme mostrado na **Figura 13**.

Uma vez selecionada a opção, a estrutura do Quick Edit é aberta (**Figura 14**) provendo um acesso rápido à configuração das principais propriedades daquele controle, evitando que o desenvolvedor necessite acessar o Object Inspector.

Suporte ao deploy de aplicações para a Windows 10 Store

Historicamente, o Delphi, desde sua concepção, manteve sua excelência na construção de aplicativos para a plataforma Windows. Logo, boa parte da comunidade tem como base esse tipo

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

de desenvolvimento, o que torna as aplicações desktop um pilar bastante sólido no contexto da ferramenta. Todavia, mediante o impacto causado no mercado pelo próprio avanço tecnológico, novas frentes de desenvolvimento tornaram-se necessárias. Neste ponto surgiram na ferramenta os recentes suportes às tecnologias mobile — iOS e Android.

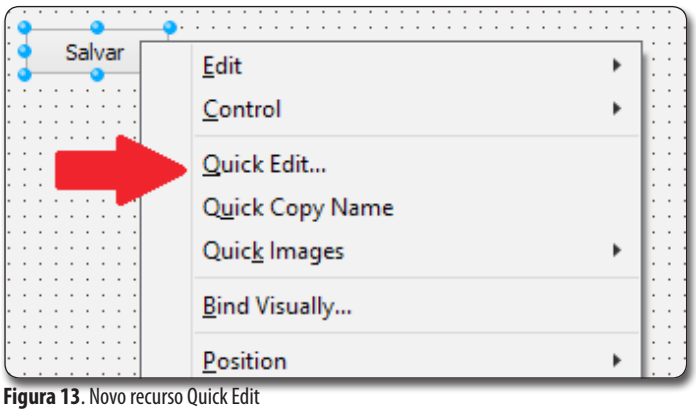

Figura 13. Novo recurso Quick Edit

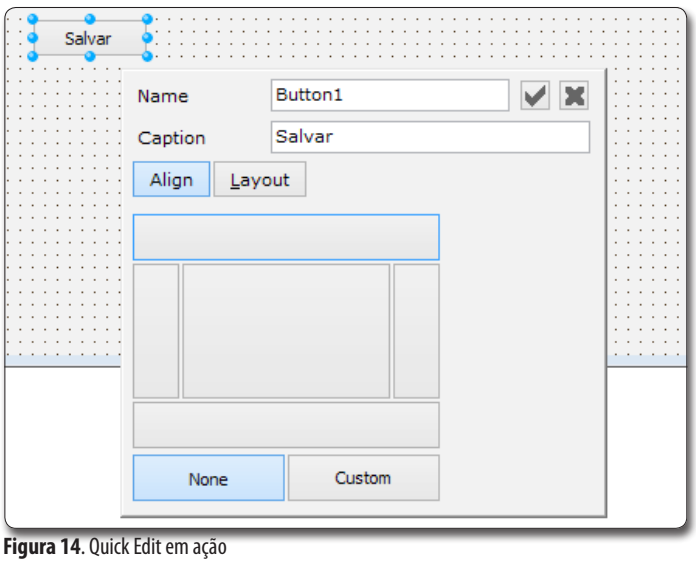

Figura 14. Quick Edit em ação

De forma natural, o desenvolvimento desktop acabou de certa forma sendo abalado por essas novas diretrizes, numa alusão a um futuro sem grandes perspectivas. No entanto, com o lançamento da mais recente versão do sistema operacional da Microsoft, o Windows 10, um novo horizonte se abre para o desenvolvimento desktop no Delphi. Indo direto ao ponto, com o Berlin Update 2, agora é possível disponibilizar uma aplicação Windows Desktop desenvolvida no Delphi na loja oficial de aplicativos do Windows 10, a Windows Store.

Tudo isso é possível graças a uma nova tecnologia suportada pelo Delphi Berlin Update 2, que permite que se gere facilmente a partir do próprio IDE os arquivos .appx necessários para o deploy de aplicações na Windows Store. A referida tecnologia é denominada Microsoft Desktop Bridge (**BOX 1**).

BOX 1. Desktop Bridge

Desktop Bridge é um conjunto de tecnologias provido pela Microsoft que habilita a conversão de aplicações desktop tradicionais (VCL Applications, Windows Forms Applications, WPF Applications, etc.) para o formato previsto pelo Windows 10 — aplicações UWP (Universal Windows Platform). Durante esse processo a aplicação desktop é empacotada em um package .appx, destinado ao Windows 10.

Windows 10 SDK

De forma resumida, conforme já citado anteriormente, para que as aplicações Delphi sejam disponibilizadas na loja oficial de aplicativos da Microsoft, se faz necessária a criação de pacotes do tipo .appx. Logo, para que isso seja possível, é mandatório que o ambiente de desenvolvimento apresente o Windows 10 SDK instalado.

Tal como já mostrado, a instalação do Update 2 do Delphi Berlin contempla a instalação desse SDK, evitando que o desenvolvedor tenha que baixá-lo e instalá-lo separadamente. Todavia, assim como acontece para o SDK do Android, para a efetiva construção de aplicativos para o Windows 10 é necessário que se adicione o SDK correspondente no IDE. Isso se dá por meio da opção *SDK Manager*, disponível no menu *Tools* > *Options* > *Environment Options* (**Figura 15**).

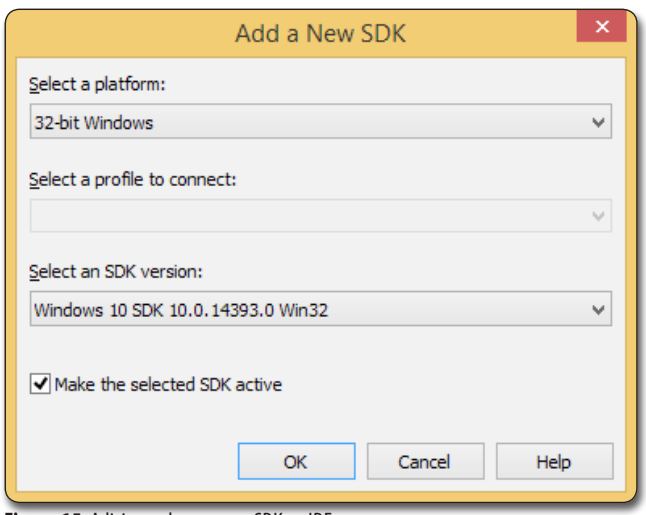

Figura 15. Adicionando um novo SDK ao IDE

Assim como pode ser visto na imagem, por padrão, o Windows 10 SDK instalado junto ao Delphi é automaticamente pré-selecionado no campo de seleção do SDK version. Todavia, caso isso não ocorra ou eventualmente se deseje utilizar uma nova versão do próprio SDK, basta usar o assistente disponibilizado para a configuração.

Esse assistente é denominado "Create a new Windows 10 SDK" e tem por função permitir que se aponte, de forma manual, os diretórios tanto da versão do SDK que se deseja utilizar quanto das diversas ferramentas essenciais e necessárias ao desenvolvimento das aplicações. A **Figura 16** mostra a tela inicial do assistente, que já busca pelo diretório default do kit de desenvolvimento do Windows 10 na máquina.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

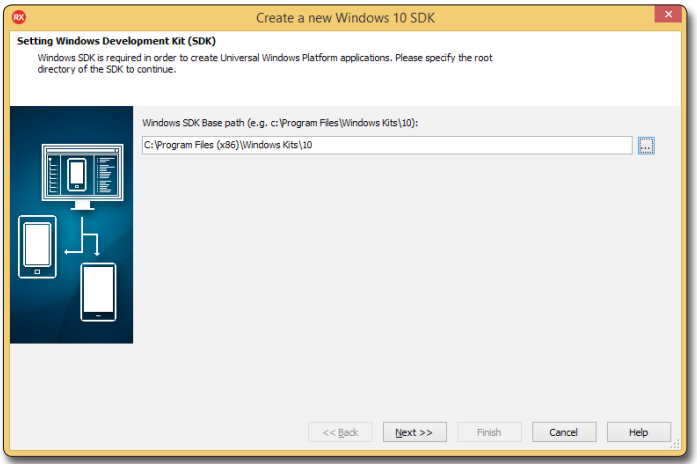

Figura 16. Assistente para a adição manual do Windows 10 SDK

Uma vez adicionado o Windows 10 SDK ao IDE, o ambiente se torna apto a construção de aplicativos para essa plataforma. Prova disso é a habilitação de uma nova opção de configuração, denominada *Application Store*, presente no *Project Manager* dentro de *Target Platforms* (veja o destaque da **Figura 17**). Assim, essa deverá ser a opção a ser selecionada quando a plataforma de destino for a Windows 10 Store.

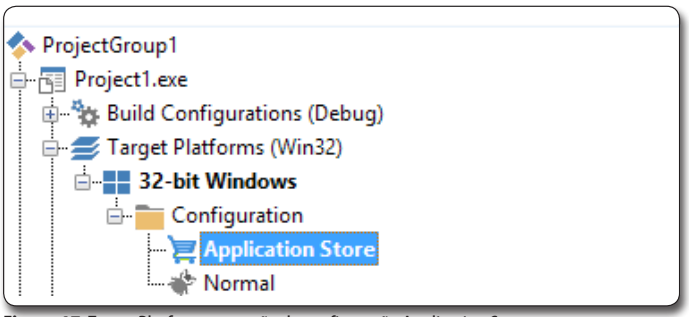

Figura 17. Target Platform — opção de configuração Application Store

Opções de projeto exclusivas para aplicações Windows 10

Com a adição do suporte a deploy de aplicações Delphi para a Windows Store, outras adições foram feitas no IDE para que esse suporte seja completo e real. Exemplo disso são as opções de projeto *Application* e *Provisioning*, que agora ganham novas adições, exclusivas a esse cenário.

Application

Entre os requisitos necessários para a distribuição da aplicação está a configuração da imagem que irá representá-la. Nesse caso, essa representação se dá por meio de logos, um menor (44x44px) e outro maior (150x150px), exclusivos a esse tipo de aplicação (**Figura 18**). Vale ressaltar que, por padrão, a logo do Delphi é atribuído e poderá ser utilizado.

Ainda de acordo com a imagem é possível notar outro detalhe que se faz presente, denominado *Manifest File*, que está relacionado à configuração do arquivo de manifesto (*AppManifest.xml*). Por padrão, esse arquivo é gerado automaticamente pelo IDE em projetos com Target Platform Windows, com fins de distribuição e deploy. Em suma, entenda esse arquivo como sendo um requisito que deverá estar presente no package de distribuição de sua aplicação. Num cenário inicial, o próprio IDE se encarrega de produzir e alimentar esse arquivo com as informações necessárias, livrando o desenvolvedor de qualquer intervenção manual.

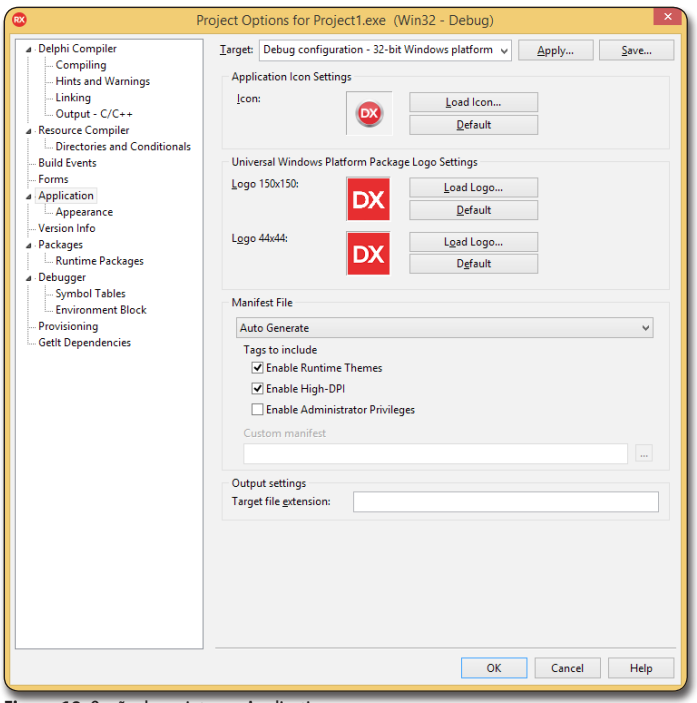

Figura 18. Opção de projeto — Application

Provisioning

Ainda com relação à distribuição de aplicativos Delphi para a Windows Store, temos a criação de um certificado da própria aplicação. Isso se faz necessário para que seja possível realizar testes desse tipo de aplicação tanto na máquina de desenvolvimento atual quanto em outro ambiente que possua o Windows 10. Aqui, entenda certificado como um arquivo especial que será gerado e estará atrelado a uma senha definida.

Essencialmente, há dois tipos de distribuição possíveis neste cenário. A primeira é denominada "Ad hoc", que condiz com uma distribuição aberta e indireta, ou seja, não direcionada à plataforma alvo, neste caso, a Windows Store.

A segunda é atribuída como "Store" e é voltada exclusivamente à submissão do aplicativo desenvolvido para a Store. Toda essa atividade é provida pelo próprio IDE em nível de projeto, através da opção *Provisioning* (**Figura 19**).

Observando o que é mostrado na imagem, fica nítido que o certificado em questão só é mandatório para uma distribuição Ad hoc. Já para uma distribuição direta à Windows Store, nenhum certificado é exigido.

Neste ponto é importante ressaltar que esse tipo de desenvolvimento exige um artigo completo, evidenciando todos esses aspectos de forma mais detalhada, tanto na teoria quanto na prática.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

O objetivo aqui é dar uma visão geral das novidades dessa versão, o que se restringe a uma amostragem panorâmica dessas nuances.

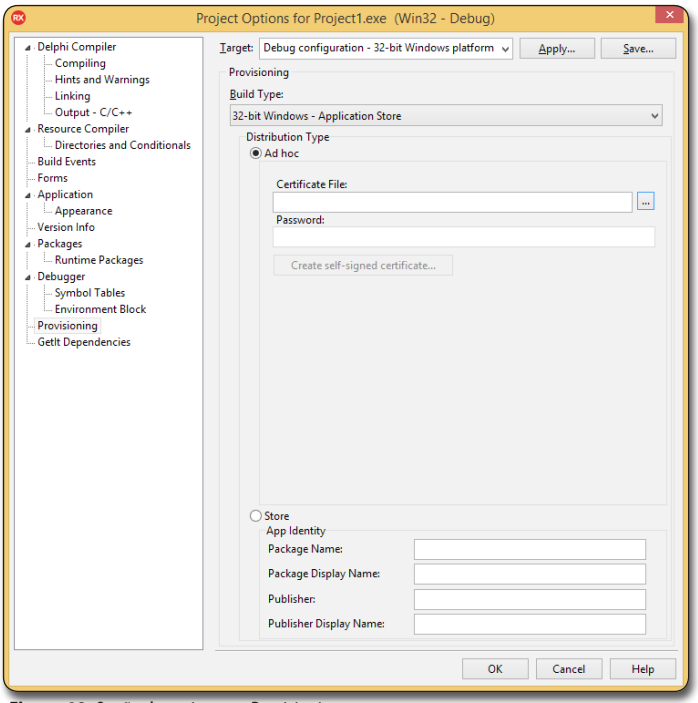

Figura 19. Opção de projeto — Provisioning

O que vem por aí

Mediante a toda essa gama de novidades trazidas pelo Delphi, uma pergunta inevitável a se fazer é: o que esperar de seu futuro? Pensando nisso, é comum as grandes fabricantes definirem de tempos em tempos o Roadmap do produto, ou seja, um roteiro oficial listando as grandes novidades que estão por vir. Ainda neste ano a IDERA/Embarcadero divulgou o Roadmap de 2017 para o Delphi, que inclui as seguintes adições:

- Suporte a servidores Linux;
- Melhorias gerais de UI e UX;
- Incorporação da biblioteca de componentes Konopka Controls para VCL e FMX;
- Incorporação da biblioteca de componentes Radiant Shapes para VCL e FMX;
- Capacidade de multi-tenancy para o RAD Server;
- Suporte a Z-Order em aplicações FMX para Android.

Entre as novidades listadas, certamente a que mais chama a atenção é o previsto suporte ao Linux, recurso que sempre foi muito pedido pela comunidade, desde a extinção do Kylix (**BOX 2**) na década passada. Além disso, isso significará a inclusão de mais uma plataforma ao leque de plataformas suportadas pela ferramenta.

Konopka Controls e Radiant Shapes são bibliotecas de terceiros, provenientes da empresa Raize Software, a mesma detentora do CodeSite. A partir de um tratado comercial, ambas as bibliotecas foram adquiridas pela Embarcadero ainda em 2015 e em 2017 passarão a integrar definitivamente a instalação nativa do IDE. Atualmente são distribuídas de forma complementar.

BOX 2. Kylix

Kylix é um ambiente de desenvolvimento integrado, criado pela Borland, voltado à linguagem de programação Object Pascal. Considerado um coirmão do IDE Delphi, o Kylix apresenta uma biblioteca de desenvolvimento própria e singular, denominada CLX, uma espécie de versão Linux da VCL. Tempos depois de sua criação, a ferramenta foi descontinuada. Nos dias atuais, o Lazarus é o ambiente de desenvolvimento integrado que mais se equivale às suas pretensões.

Konopka Controls

A biblioteca Konopka Controls era anteriormente conhecida como Raize Components, tendo sido criada há mais de uma década pela Raize Software. Essa longevidade dá mostras claras da maturidade que o produto possui, o que acaba por justificar a sua aquisição. Sua estrutura é composta por mais de uma centena de controles VCL, com suporte a 64 bits e estilos (VCL Styles). Sob o domínio da Embarcadero, a biblioteca foi rebatizada para Konopka Signature VCL Controls, sendo então definida oficialmente da seguinte forma:

"Os controles Konopka Signature VCL permitem o design rápido de interfaces de usuário modernas e sofisticadas para aplicativos Windows. Com um conjunto de mais de 200 controles de interface do usuário do Windows, projetados para tornar as aplicações mais intuitivas e visualmente impressionantes, os desenvolvedores do Delphi e C ++ Builder podem modernizar e simplificar todas as experiências de usuário do Windows. Com o suporte completo a VCL Styles, os desenvolvedores podem oferecer aplicativos personalizados e modernos do Windows 10 com facilidade".

Obviamente, a explanação ideal da biblioteca exige um artigo exclusivo, todavia, apenas para citar, a seguir são enumerados alguns de seus principais componentes:

- TRzEdit: controle Edit, similar ao tradicional TEdit, porém com ajustes complementares, tal como o alinhamento de conteúdo à direita;
- TRzButtonEdit: controle Edit com suporte a dois botões incorporados tal qual o TButtonedEdit da VCL;
- TRzPanel: componente Panel com recursos aprimorados de exibição, incluindo um Dock Manager particular;
- TRzSizePanel: componente Panel com suporte a redimensionamento em tempo de execução;
- TRzCheckGroup: componente GroupBox que cria e gerencia automaticamente um conjunto de controles CheckBox;
- TRzDBRadioGroup: componente com ligação a dados que cria e gerencia automaticamente um conjunto de controles RadioButton;
- TRzButton: controle Button com capacidades de Caption multilinha, estilização de texto 3D e cores personalizadas;
- TRzRapidFireButton: controle Button que dispara repetidamente o evento de clique enquanto o botão é mantido pressionado;
- TRzGroupController: componente não visual que provê um ponto único de alteração da aparência de um grupo definido de controles;

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

• TRzURLLabel: controle Label com suporte à criação de hiperlinks para páginas web, e-mails ou arquivos.

Indo além

Além das novidades trazidas de forma nativa pela ferramenta, outras inúmeras surgem a partir de empresas externas. Isso dá origem aos recursos 3rd-Party, ou recursos de terceiros, como se costuma dizer no popular. Tais empresas acabam então por acompanhar a evolução da própria ferramenta principal, neste caso o Delphi, lançando novas versões de suas bibliotecas, componentes e ferramentas a cada novo release do IDE. Em vista disso, é de suma importância o desenvolvedor Delphi se manter antenado também a esse tipo de cenário, uma vez que os recursos de terceiros são produzidos essencialmente para resolver algum aspecto do desenvolvimento de software, em complemento aos recursos da própria ferramenta principal.

IDE Palette Menu

Em meio as inúmeras empresas desse ramo, a já citada Raize Software é um exemplo, e atualmente disponibiliza vários produtos relacionados ao Delphi Berlin. Entre esses produtos está a ferramenta IDE Palette Menu, que simplifica o acesso às diversas guias da Tool Palette, conforme mostra a **Figura 20**.

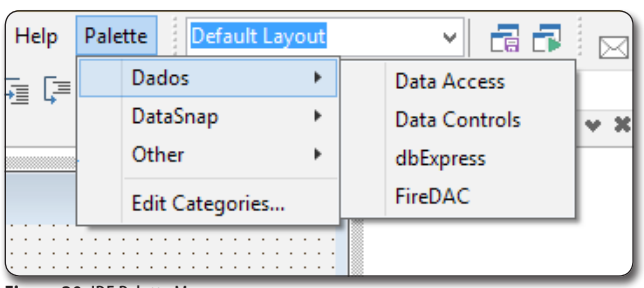

Figura 20. IDE Palette Menu

Na prática, uma vez instalada, a ferramenta toma forma numa nova opção na barra de ferramentas do IDE, possibilitando que se categorize o acesso às diversas guias da paleta de componentes. Isso evita a ação de scroll pela busca de determinada guia, simplificando o acesso. Como pode ser observado na imagem, foi definida uma categoria "Dados", com acesso rápido às guias Data Access, Data Controls, dbExpress e FireDAC.

TMS Aurelius

Outro exemplo de empresa que fornece uma série de soluções complementares ao Delphi, sempre acompanhando sua evolução de lançamentos, é a TMS Software. Entre seu leque de opções estão bibliotecas para VCL, FMX e IntraWeb, além de ferramentas voltadas a business, análise e desenvolvimento, todas elas já com versões disponíveis para o Delphi Berlin 10.1. Em meio a essas opções, encontra-se o TMS Aurelius, que, desde certo tempo, vem ganhando cada vez mais notoriedade em meio à comunidade.

Em suma, o TMS Aurelius é um framework de mapeamento objeto-relacional. Essa denominação vem do termo em inglês Object-Relational Mapping (ORM), que é uma técnica de desenvolvimento empregada com o objetivo de reduzir a complexidade de trabalhar com bancos de dados relacionais quando estamos programando orientado a objetos.

Na prática, ao se utilizar o Aurelius (como é popularmente chamado), o desenvolvedor se torna capaz de manipular seus objetos em plena concordância com uma base de dados comum (ex: Oracle, SQL Server, Firebird, etc.). Isso porque ao "salvar" um objeto de negócio numa tabela, por exemplo, cabe ao Aurelius fazer a interpretação dessa entidade, transformando-a em um formato de dado legível ao banco. Do contrário, ainda nesse cenário, a ausência de um framework igual ao Aurelius implicaria numa codificação manual por parte do próprio desenvolvedor, ação essa que acaba por desencorajar muitos, dada sua complexidade e baixa produtividade.

Naturalmente, o TMS Aurelius não se restringe à persistência de objetos, contemplando todos os benefícios que podem ser obtidos por meio do uso de um framework ORM. Um exemplo disso é a consulta de dados numa aplicação, que, tradicionalmente, se dá por meio do uso de comandos SQL (**Listagem 3**).

Listagem 3. Exemplo de uma consulta a dados feita na aplicação

```
01 FDQuery1.Close;
02 FDQuery1.SQL.Clear;
03 FDQuery1.SQL.Add('select');
04 FDQuery1.SQL.Add('EMP_NO, FULL_NAME, HIRE_DATE');
05 FDQuery1.SQL.Add('from EMPLOYEE');
06 FDQuery1.SQL.Add('where JOB_COUNTRY = :COUNTRY');
07 FDQuery1.ParamByName('EMP_NO').AsInteger := 1;
08 FDQuery1.Open;
```

Essa listagem exibe um trecho de código em que se faz uma consulta aos dados da tabela EMPLOYEE do banco EMPLOYEE do Firebird, selecionando-se os campos EMP_NO, FULL_NAME e HIRE_DATE. A intenção aqui é exibir as informações do empregado de número 1 (EMP_NO = 1). Numa aplicação orientada a objetos, com o uso do TMS Aurelius, essa mesma consulta poderia ser resumida a uma única linha de código:

```
Employee = Manager1.Find<TEmployee>(1);
```

Em complemento, a seguir são mostrados dois trechos de código. O primeiro, na **Listagem 4**, mostra a codificação básica de uma classe simples no Delphi, denominada *TPessoa*. Tradicionalmente, essa estrutura representaria uma entidade no sistema, sendo composta, então, pelos atributos que lhe são peculiares, tais como *Id* (referência), *Nome*, *Idade*, *Sexo* e *Ativo* (se está ativa no sistema).

Já no trecho de código mostrado na **Listagem 5**, tem-se a mesma classe, *TPessoa*, porém agora devidamente mapeada para o contexto do Aurelius. Nota-se, então, que somente dois pontos foram acrescidos com relação ao trecho anterior, na realidade dois atributos do framework: [Entity] e [Automapping]. O primeiro indica ao Aurelius que essa classe é uma entidade que pode ser persistida no banco de dados, enquanto que o segundo é uma

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

marcação especial que estabelece o auto mapeamento de todas as propriedades contidas na classe. Em outras palavras, por convenção, o próprio framework se encarregará de relacionar, por exemplo, a property *Nome* da classe *TPessoa* com o campo *Nome* da tabela *Pessoa*.

Listagem 4. Uma classe TPessoa

```
01 TPessoa = class
02  private
03    FAtivo: Boolean;
04    FId: Integer;
05    FIdade: Integer;
06    FSexo: string;
07    FNome: string;
08    procedure SetAtivo(const Value: Boolean);
09    procedure SetId(const Value: Integer);
10    procedure SetIdade(const Value: Integer);
11    procedure SetNome(const Value: string);
12    procedure SetSexo(const Value: string);
13  public
14    property Id: Integer read FId;
15    property Nome: string read FNome write FNome;
16    property Idade: Integer read FIdade write FIdade;
17    property Sexo: string read FSexo write FSexo;
18    property Ativo: Boolean read FAtivo write FAtivo;
19 end;
```

Listagem 5. Classe TPessoa mapeada com o TMS Aurelius

```
01 [Entity]
02 [Automapping]
03  TPessoa = class
04  private
05    FAtivo: Boolean;
06    FId: Integer;
07    FIdade: Integer;
08    FSexo: string;
09    FNome: string;
10    //setters omitidos
11  public
12    property Id: Integer read FId;
13    property Nome: string read FNome write FNome;
14    property Idade: Integer read FIdade write FIdade;
15    property Sexo: string read FSexo write FSexo;
16    property Ativo: Boolean read FAtivo write FAtivo;
17  end;
```

A plena utilização de um ORM no desenvolvimento Delphi, assim como citado quando falamos de outras tecnologias neste artigo, exige uma série dedicada sobre o tema, uma vez que vários são os pontos a serem abordados. O próprio Aurelius, apesar de simples e produtivo, acaba por requerer um estudo mais aprofundado, principalmente no estágio inicial de aprendizagem.

Definitivamente esse é o momento de tirar proveito do melhor que a ferramenta proporciona: desenvolvimento nativo e multiplataforma, com toda a produtividade RAD já conhecida. A partir de um único ambiente integrado, você poderá desenvolver diversos tipos de aplicações — mais clássicas (Windows 32 e 64 bits e MacOS), móveis (iOS e Android) ou mesmo com um passo no futuro (Windows 10) — sem enrolações e com um reaproveitamento de conhecimento que só o Delphi proporciona (mesma linguagem e mesma base de componentes).

Agora, cabe a você, caro amigo desenvolvedor, literalmente pôr as mãos na massa e, a partir de uma mesma base de código, construir suas aplicações para as principais plataformas do mercado. Espero que vocês tenham gostado do artigo e nos vemos na próxima. Bons desenvolvimentos!

Autor

Fabrício Hissao Kawata
fabriciohk@hotmail.com
Formado em Tecnologia em Análise e Desenvolvimento de Sistemas e pós-graduado em Engenharia de Componentes Utilizando Java. Analista e Consultor Delphi com 10 anos de experiência. Atuou como Instrutor Oficial Embarcadero, na Kees Informática. Certified Delphi Developer e Certified Embarcadero Instructor.

Links:

Delphi – Página oficial do produto
https://www.embarcadero.com/br/products/delphi

Delphi Berlin Update 2 – Página de download da versão de avaliação (Trial)
https://www.embarcadero.com/br/products/delphi/start-for-free

Delphi 10.1 Berlin Starter Edition (versão gratuita da ferramenta)
https://www.embarcadero.com/br/products/delphi/starter/promotional-download

Raize - IDE Palette Menu
https://www.raize.com/DevTools/Tools/PaletteMenu.asp

TMS Aurelius
http://www.tmssoftware.com/site/aurelius.asp

Lista completa de componentes do Konopka Controls
https://www.embarcadero.com/products/konopka-signature-vcl-controls/component-list

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

IntraWeb: Como criar uma aplicação web passo a passo

Aprenda a desenvolver aplicações para a web com Delphi

Por muito tempo, as aplicações desktop dominaram o mercado de software e atenderam perfeitamente a maior parte das necessidades, desde simples formulários de cadastro e relatórios, até sistemas complexos de automação e módulos de "frente de loja"/PDV. No entanto, com o passar do tempo e o avanço da internet, surgiu para muitos usuários a necessidade de ter seus sistemas disponíveis online, de forma que pudessem acessá-los de diferentes locais e de forma independente de plataforma.

No Delphi, o desenvolvimento de aplicações para Windows foi um dos fatores que levaram ao sucesso dessa ferramenta, no entanto, quando nos deparamos com a necessidade de criar aplicações web, muitos programadores optam por outras linguagens, como Java, PHP e C#. Em boa parte dos casos, essa escolha ocorre pelo não conhecimento das opções disponíveis no Delphi para atender a essa demanda, entre as quais destaca-se o IntraWeb, que será foco deste artigo.

O IntraWeb, que atualmente encontra-se na versão XIV, é um framework mantido pela Atozed Software e distribuído juntamente ao Delphi pela Embarcadero, com o objetivo de permitir e simplificar o desenvolvimento de aplicações web. Com ele, é possível desenvolver sistemas completos para a web, integrando frameworks e bibliotecas comuns, como Bootstrap, para melhorar a interface gráfica das aplicações e torná-las totalmente adequadas a esse ambiente.

Ao desenvolver um projeto com esse framework, utiliza-se apenas componentes visuais das paletas *IntraWeb*, devido ao fato de eles serem renderizados como HTML. Esses componentes estão dispostos em quatro paletas:

Fique por Dentro

Neste artigo veremos como é possível desenvolver aplicações web no Delphi utilizando o framework IntraWeb, assim como quais as vantagens e desvantagens de desenvolver e manter uma aplicação com esse framework. Além disso, veremos como é possível utilizar frameworks como o Bootstrap e o Materialize a fim de melhorar o visual da aplicação. Esse tema é útil para todo desenvolvedor Delphi que deseja construir aplicações web e não pretende adotar outra linguagem de programação.

- *IntraWeb Standard*: Semelhante à paleta *Standard* da VCL (*Visual Component Library*), nela estão localizados os componentes mais simples, como labels, buttons, comboboxes, grids, checkboxes, menus, além de componentes específicos do ambiente web, como links, applets, flash, layers, etc.;
- *IntraWeb Data*: Semelhante à paleta *Data Controls* da VCL, nessa paleta estão os componentes visuais para manipulação da base de dados, como DBGrid, DBLabel, DBEdit e DBNavigator, que, assim como os da VCL, funcionam ligados a um dataset;
- *IntraWeb Control*: Aqui estão os componentes utilizados no controle da página, como layouts (templates) e produtores de páginas;
- *IntraWeb Authentication*: Nessa paleta estão disponíveis componentes para gerenciamento da política de segurança.

Neste artigo vamos desenvolver um projeto completo com IntraWeb, a fim de explorar seus principais conceitos e componentes.

Modelagem do projeto

O projeto que desenvolveremos aqui será uma aplicação que permitirá que os clientes efetuem pedidos online, funcionando como um módulo online de um ERP (*Enterprise Resource Planning*).

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Para isso, teremos as telas de login; produtos disponíveis, na qual o cliente poderá adicionar ao carrinho os produtos desejados; lista dos itens no carrinho; tela de gerenciamento dos endereços do cliente logado; e uma página para confirmar a forma de pagamento e endereço de entrega desejado. O projeto será baseado no modelo de banco de dados mostrado na **Figura 1**, que, para este artigo, foi implementado em um banco de dados MySQL chamado de *intrawebapp*.

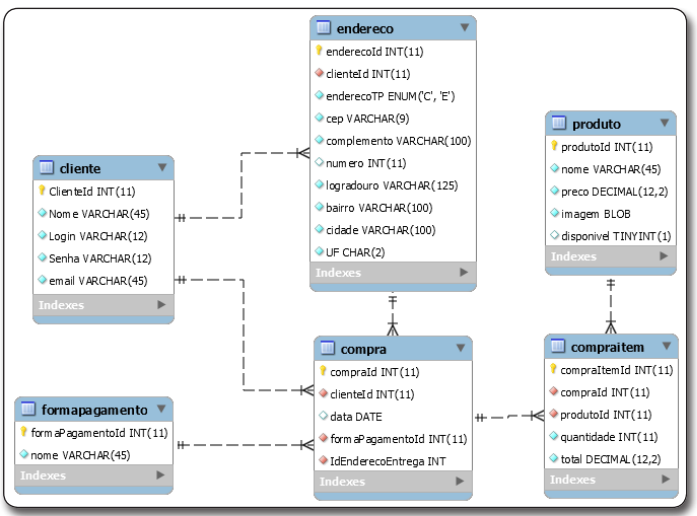

Figura 1. Modelo Entidade Relacionamento do banco de dados

Esse módulo será de uso exclusivo do cliente, portanto não contará com o cadastro de clientes ou produtos, apenas será responsável pelo login e venda dos itens. Assumimos então que os demais módulos do ERP são responsáveis por essas e outras funcionalidades comuns nesse tipo de sistema.

Criando o projeto

Para criar uma aplicação IntraWeb, vá ao menu *File* > *New* > *Other*, e na aba *IntraWeb* (presente na pasta *Delphi Projects*) selecione *IntraWeb Application Wizard*. No wizard selecione a opção *Stand Alone Application*, digite o nome do seu projeto (para este artigo usaremos "AppIntraWeb") e, em *Project Directory*, defina o diretório do projeto.

Nota

Aplicações IntraWeb podem ser hospedadas tanto em servidores Windows quanto Linux. Para hospedagem em servidores Windows, pode-se utilizar aplicações Stand Alone, nas quais não se faz necessário o uso de serviços como ISS ou Apache. Para hospedagem em um servidor ISS ou Apache é necessário que a aplicação tenha sido criada como ISAPI.

A primeira unit a ser criada será a tela de login da aplicação, que deve ser alterada para que se pareça com a apresentada na **Figura 2**.

A tela de login é composta por um *IWRegion*, e dentro dele estão os componentes de imagem (*TIWImage*), label, edit e button, que será chamado de *btnEntrar*. O label em vermelho, que receberá o nome *lblAviso*, apresentará a mensagem de erro caso o usuário entre com as informações de login incorretas, por isso ele deve ter sua propriedade *visible* configurada como *false* (a sua visibilidade será alterada na tentativa do login).

Figura 2. Tela de login

Para o login será preciso acessar a tabela *cliente* do nosso banco de dados, então é necessário que uma conexão seja configurada. Abra a *UserSessionUnit*, renomeie-a para *UserSession* e adicione os componentes *TFDConnection* e *TFDPhysMySQLDriverLink*. O *TFDConnection* é responsável pela conexão com o banco de dados e o *TFDPhysMySQLDriverLink* é responsável por disponibilizar o driver de conexão com o MySQL para o *TFDConnection* (caso ocorram erros de conexão, verifique a propriedade *VendorLib* do *TFDPhysMySQLDriverLink*). Altere o nome do componente de conexão para *Connection* e, com um clique com o botão direito do mouse, selecione a opção *Connection editor*. Será aberta uma janela para configurar a conexão com o banco de dados, como mostra a **Figura 3**. Selecione o *mySQL* como *Driver ID* e configure as propriedades de conexão.

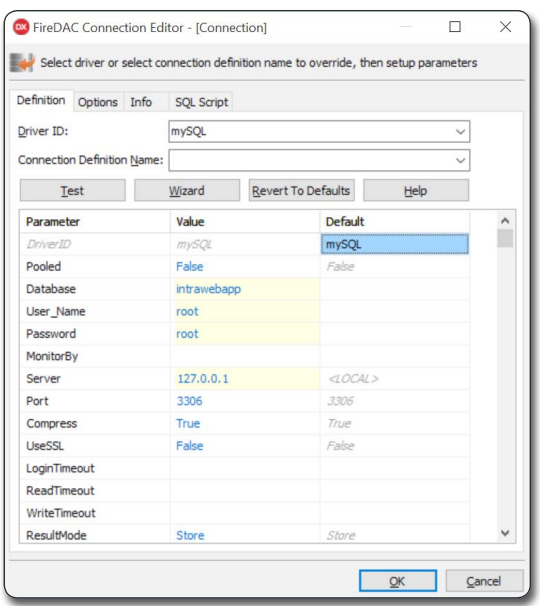

Figura 3. Connection Editor

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

A cada nova requisição à aplicação (novo login) será instanciado um novo *UserSession*, que é de conhecimento apenas da sessão. Com isso, tudo que se desejar controlar referente à sessão deve ser definido nessa unit. Já para conteúdo global deve-se utilizar a *ServerController*.

Em seguida, adicione a propriedade *idUsuario*, do tipo *integer*, na classe *UserSession*. Essa propriedade será preenchida com o id do usuário após a realização do login, sendo possível, assim, sabermos qual usuário está logado em qualquer momento da nossa aplicação.

Com a conexão configurada e com a propriedade *idUsuario* inserida, retorne ao formulário de login, adicione a ele uma *TFDQuery* e renomeie-a para *qryLogin*. Depois, edite sua propriedade SQL para que fique como a **Listagem 1**.

Nesse código SQL é feita uma consulta à tabela *cliente*, buscando pelo registro cujos login e senha são iguais aos parâmetros passados. Para validar o login crie, como private, a função *validarLogin*, que será responsável por checar se o login está correto ou não, e retornar o id do usuário que pretende entrar na aplicação. Implemente esse método como na **Listagem 2**.

Listagem 1. Comandos SQL para validar o login

```
01 SELECT *
02 FROM CLIENTE
03 WHERE LOGIN =:LOGIN
04 AND   SENHA =:SENHA
```

Listagem 2. Método validarLogin

```
01 function TfrmLogin.validarLogin(psLogin, psSenha: String): Integer;
02 begin
03   try
04     qryLogin.Close;
05     qryLogin.Connection := UserSession.Connection;
06     qryLogin.ParamByName('login').AsString := psLogin;
07     qryLogin.ParamByName('senha').AsString := psSenha;
08     qryLogin.Open();
09
10     result := qryLogin.FieldByName('clienteId').AsInteger;
11   finally
12     qryLogin.Close
13   end;
14 end;
```

Esse método recebe por parâmetro o usuário e a senha que se deseja validar e repassa esses valores para a *qryLogin*. Na linha 5 é definido que a conexão para a *qryLogin* será a mesma criada em *UserSession*. Depois de os parâmetros serem passados para a query (linhas 6 e 7), o método *open* é chamado para abri-la (linha 8) e o *clienteId* é salvo na variável *result* (linha 10).

O próximo passo será implementar o evento *OnClick* do botão de login — veja o código presente na **Listagem 3**.

Nesse código, na linha 4, a variável *idUsuario* recebe o retorno do método *validarLogin*. Já na linha 5 é testado se o *idUsuario* está vazio: caso esteja, esse usuário não foi encontrado na base dados ou a senha informada está errada, e, por consequência, o *lblAviso* será mostrado e o restante do processamento será abortado.

Listagem 3. Evento OnClick do botão btnEntrar da tela de login

```
01 procedure TfrmLogin.btnEntrarClick(Sender: TObject);
02 Var idUsuario : Integer;
03 begin
04   idUsuario := validarLogin(edtUsuario.Text, edtSenha.Text);
05   if (idUsuario = 0) then
06   begin
07     lblAviso.Visible := true;
08     Exit;
09   end;
10
11   UserSession.idUsuario := idUsuario;
12   Self.Release;
13   TfrmMenu.Create(WebApplication).Show;
14 end;
```

Caso o usuário seja encontrado na base e a senha esteja correta, o *idUsuario* da *UserSession* recebe o valor do retorno do método de validação.

Em seguida, na linha 12, é feito um release do formulário atual e, na linha 13, é executado o método *show* para uma instância de *TfrmMenu*, cuja classe ainda não foi criada. Portanto, crie um novo formulário, clicando em *File* > *New* > *Other* > *IntraWeb* > *NewForm*, e renomeie-o para *frmMenu*. Esse formulário será o responsável por toda a navegação das páginas, e será dentro dele que as páginas serão apresentadas.

Adicione ao *frmMenu* um *TIWRegion* e, dentro dele, dois *TIWMenu*: um deve ter a propriedade *align* configurada como *alTop*, e o outro como *alBottom*. Entre eles adicione uma imagem e, depois, dois componentes *TMainMenu*. Um dos menus deve conter as opções Perfil, Carrinho e LogOut, e o outro, as opções Home e Produtos. Na propriedade *AttachedMenu* dos elementos *TIWMenu*, faça o link com os itens de menu para que a tela fique semelhante à da **Figura 4**.

Figura 4. Tela frmMenu

Com o menu criado, adicione ao evento *OnClick* da opção *LogOut* o seguinte trecho de código:

```
Self.Release;
TfrmLogin.Create(WebApplication).Show;
```

Assim como no login, o método *Release* é chamado para o formulário atual, e, em seguida, é acionado o método *Show* para o formulário que se deseja apresentar.

Compile o projeto, e uma janela como a da **Figura 5** será apresentada. A aplicação pode ser testada de duas formas: a primeira é clicando no botão com o ícone de um navegador, e a segunda é digitando, no navegador, a URL 127.0.0.1 acompanhada da porta

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

do serviço, que pode ser obtida no log do Server como indicado na **Figura 5**.

Ao executar a aplicação em seu browser, a tela de login deve ser apresentada e a função de validar deve estar funcionando, isto é, ao efetuar o login o menu será apresentado.

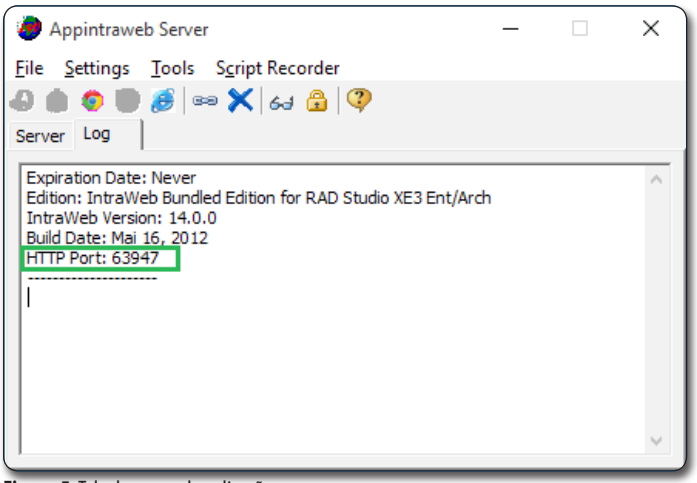

Figura 5. Tela do server da aplicação

Nota

Para as versões de avaliação do IntraWeb a porta é randômica. Aqui é importante citar que em algumas versões do Delphi o IntraWeb está sendo reconhecido como avaliação mesmo com a versão original. Para a correção desse problema, deve-se entrar no site oficial da Atozed (indicado na seção **Links**) e baixar a atualização.

Trabalhando com frames

Em uma aplicação comum, pode-se instanciar formulários dentro de outros, mas em aplicações IntraWeb isso é impossível, e, além disso, apenas um formulário ativo é permitido por vez. O que pode ser feito para todas as demais janelas da aplicação serem abertas dentro da janela de menu é instanciar frames dentro de formulários. Para criar um frame vá em *Menu* > *Other* > *IntraWeb* > *New Frame*, renomeie-o para *frmPerfil* e salve-o.

Retorne ao menu e adicione na seção *private* a seguinte variável, que será utilizada para controlar o frame ativo:

```
frameActive : TFrame;
```

Depois, crie como *public* o método *AbrirFrame*, que será o responsável por abrir o frame dentro do formulário de menu (vide **Listagem 4**).

Esse método recebe por parâmetro a classe do frame que será instanciada e, na linha 3, é verificado se já existe uma instância para *frameActive*: caso exista, é verificado se os objetos são da mesma classe para que não haja destruição e renderização desnecessárias. Depois das validações, o *frameActive* recebe uma instância do frame solicitado, o parent do frame passa a ser o formulário de menu e o alinhamento do frame passa a ser *alClient*. Por fim, a instância do frame é repassada ao *result*.

Listagem 4. Método AbrirFrame

```
01 function TfrmMenu.AbrirFrame(poFrameClass: TCustomFrameClass) : TFrame;
02 begin
03   if Assigned(frameActive) then
04   begin
05     if (frameActive.ClassType = poFrameClass) then
06       Exit;
07
08     FreeAndNil(frameActive);
09   end;
10
11   frameActive      := TFrame(poFrameClass.Create(Self));
12   frameActive.Parent := self;
13   frameActive.Align  := alClient;
14
15   result := frameActive;
16 end;
```

O método *AbrirFrame* evita que todos os testes tenham que ser repetidos a cada novo frame desejado, evitando, assim, a redundância de código. Sempre que for necessário criar um frame para um menu, basta executar o comando *"AbrirFrame(<classe_frame_desejada>);"*. Portanto, na tela de menu, adicione o comando *"AbrirFrame(TfrmPerfil);"* ao evento *OnClick* do menu *perfil*.

Como nada foi implementado no *frmPerfil*, tudo ficará branco, dificultando a identificação de seu funcionamento. Então, para ajudar na identificação, altere a cor de fundo do *frmPerfil* apenas para fins de teste. Em seguida, compile o projeto e confira se o menu *Perfil* está funcionando corretamente.

Elaborando um CRUD

Retorne ao *frmPerfil*, que será a tela responsável por apresentar as informações do cliente e permitir consultar, incluir, editar e excluir seus endereços, e edite-o conforme a **Figura 6**.

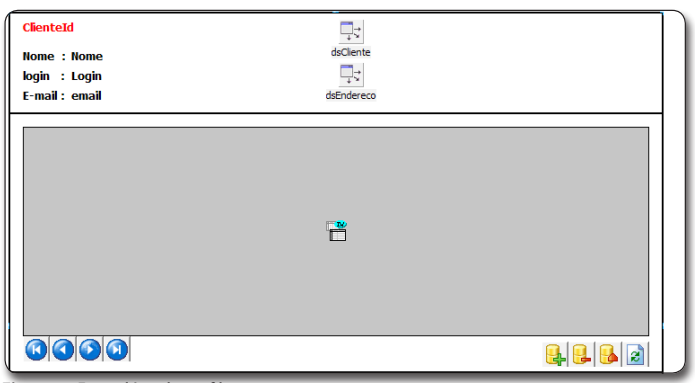

Figura 6. Formulário de perfil

A tela de perfil está dividida em duas regions: na primeira serão apresentados os dados do cliente com a utilização de componentes *TIWLabel* e *TIWDBLabel* para os títulos e valores carregados do banco, respectivamente. A segunda region é composta por um *TIWDBGrid* e dois componentes *TIWDBNavigator*, sendo que no grid serão apresentados os endereços do cliente, e os navigators irão controlar as ações no dataset referente à grid. Note que o navigator do canto esquerdo irá apenas controlar a navegação

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

entre os registros e o do canto direito irá controlar as ações de insert, edit, delete e refresh.

Nessa tela será necessário consultar as tabelas de cliente e endereço, para isso retorne ao *UserSession* e adicione duas queries: a *qryCliente*, cujo código SQL encontra-se na **Listagem 5**, e a *qryEndereco*, com o código da **Listagem 6**.

A consulta do cliente é bem simples. Nela é feito um select na tabela de cliente retornando os registros cujo *clienteId* é igual ao valor do parâmetro.

Listagem 5. Instrução SQL para consulta de dados do cliente

```
01 SELECT *
02 FROM CLIENTE
03 WHERE CLIENTEID =:CLIENTEID
```

Listagem 6. Instrução SQL para retornar os endereços do cliente

```
01 SELECT
02   CASE ENDERECOTP
03     WHEN 'E' THEN 'ENTREGA'
04     ELSE 'CORRESPONDENCIA'
05   END  AS ENDERECOTPFORMATADO,
06   ENDERECO.*
07 FROM ENDERECO
08 WHERE CLIENTEID =:CLIENTEID
```

A consulta de endereços se torna um pouco mais complexa devido à implementação do *CASE* da linha 2, no qual é verificado se o *EnderecoTP* é igual a "E" ou "C". Para cada tipo é retornado uma string (“Entrega” ou “Correspondência”, respectivamente), identificada, como mostra a linha 5, como *EnderecoTPFormatado*. Após isso, os campos da tabela de endereço que cumprem à regra da linha 8 (*clienteId* igual ao valor passado por parâmetro) são retornados.

Em seguida declare e implemente dois métodos como públicos: o *OpenQryCliente* e o *OpenQryEndereco*, que serão responsáveis por passar o parâmetro às queries e fazer suas aberturas (veja a **Listagem 7**). Os dois são muito semelhantes e devem garantir que a tabela esteja fechada para passarem o id do usuário logado para o parâmetro *clienteId* e, em seguida, executarem o *open* da query.

Retorne ao *frmPerfil*, e, no evento *OnCreate* do *IWFrameRegion*, implemente o código da **Listagem 8**, para que, ao criar o formulário, seja feita a consulta para popular os dados na tela. No código dessa listagem cada dataset é vinculado a sua query correspondente e, depois, é disparado o método responsável por abri-las.

Em seguida, assim como em uma aplicação Delphi comum, vincule a grid e os componentes DB ao seus respectivos DataSources e colunas. Para criar as colunas da grid, vá à propriedade *Columns* e adicione as colunas que deseja apresentar, ligue-as aos *Fields* do DataSource e, se desejar, altere a propriedade *Titles* > *Text* para que o título da coluna fique mais apresentável. Ainda na grid, altere a propriedade *RowCurrentColor*, que altera a cor do registro corrente, para uma cor de destaque ao seu critério.

Agora, altere os componentes *navigator* para apresentarem somente os botões desejados, tomando como base a **Figura 6**, e faça a ligação ao DataSource de endereço. Em seguida, compile a aplicação e veja que a apresentação dos dados está funcionando, mas que ainda é necessário implementar uma forma de inserir dados e editar. Retorne então ao design e adicione uma nova region ao formulário, deixando-a como a **Figura 7**.

Listagem 7. Métodos responsáveis por fazer as consultas de cliente e endereços

```
01 procedure TIWUserSession.OpenQryCliente;
02 begin
03   qryCliente.Close();
04   qryCliente.ParamByName('clienteId').AsInteger := idUsuario;
05   qryCliente.Open();
06 end;
07
08 procedure TIWUserSession.OpenQryEndereco;
09 begin
10   qryEndereco.Close();
11   qryEndereco.ParamByName('clienteId').AsInteger := idUsuario;
12   qryEndereco.Open();
13 end;
```

Listagem 8. OnCreate do IWFrameRegion do frmPerfil

```
01 procedure TfrmPerfil.IWFrameRegionCreate(Sender: TObject);
02 begin
03   dsCliente.DataSet  := UserSession.qryCliente;
04   UserSession.OpenQryCliente();
05
06   dsEndereco.DataSet := UserSession.qryEndereco;
07   UserSession.OpenQryEndereco;
08 end;
```

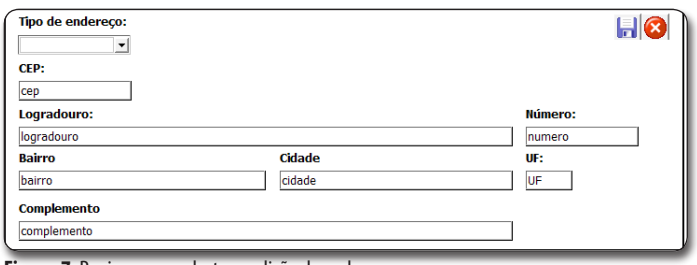

Figura 7. Region para cadastro e edição de endereço

Nesse region serão adicionados um *TIWLabel* e um *TIWDBEdit* para cada campo da tabela de endereço, além de um *TIWDBNavigator* para exibir os botões de salvar e cancelar. Faça a ligação do *TIWDBNavigator* com o DataSource de endereço, e as ligações dos componentes *TIWDBEdit* com o DataSource e os fields correspondentes. Note que o componente de tipo de endereço será um *TIWComboBox*, portanto, em sua propriedade *Items*, adicione as opções "ENTREGA" e "CORRESPONDENCIA".

Altere a propriedade *Visible* da region para *false*, pois ela deverá ser exibida apenas quando for clicado sobre o botão de inserção ou de edição. Vale citar também que, quando o botão de salvar ou cancelar for clicado, a visibilidade da region de cadastro deve ser alterada para *false* novamente. Como existirão alguns pontos que utilizarão blocos de comando semelhantes, crie um método chamado *VisibleGrid* e implemente-o como na **Listagem 9**.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 9. Implementação do método VisibleGrid

```
01 procedure TfrmPerfil.VisibleGrid(pbVisible: Boolean);
02 begin
03   pnlGridEndereco.Visible := pbVisible;
04   pnlEditEndereco.Visible := Not(pbVisible);
05 end;
```

A lógica desse método é bem simples: se o grid vai ser apresentado, o formulário de edição não será, e vice-versa. No componente navigator dos botões de CRUD, vá aos eventos *OnEdit* e *OnInsert* e chame o método criado passando *false* como parâmetro. No navigator do formulário, para os eventos *OnCancel* e *OnPost*, também faça a chamada do método *VisibleGrid*, mas dessa vez passe *true* para o parâmetro *pbVisible*.

Até então quase tudo está funcionando como o esperado. Ao clicar sobre o botão de insert é apresentado o formulário de cadastro, e, ao navegar entre os registros e clicar sobre editar, os dados são carregados corretamente, mas ainda existem dois problemas: o valor para o campo de tipo de endereço não está sendo corretamente apresentando e o seu valor não está sendo repassado para o post. Além disso, ao tentar salvar um novo registro, um erro será apresentado porque a coluna *clienteId* está nula, isto é, em nenhum momento está sendo preenchida com o valor do usuário logado. Para solucionar esse último problema, na *UserSession*, vá ao evento *AfterInsert* da *qryEndereco* e adicione a seguinte linha:

```
qryEndereco.FieldByName('ClienteId').AsInteger := idUsuario;
```

Retorne em seguida ao *frmPerfil* e no evento *OnEdit* do componente *navigatorControle* (o *TIWDBNavigator* na direita da **Figura 6**) adicione o código da **Listagem 10** abaixo da chamada do método *VisibleGrid*.

Essa implementação faz com que, ao editar um registro, seja verificado qual é o valor da coluna *enderecoTP* e o index correspondente ao valor seja repassado ao *EdtTpEndereco*.

Listagem 10. Implementação para atualizar o campo de tipo de endereco

```
01  case dsEndereco.DataSet.FieldByName('enderecoTP').AsString[1] of
02   'E' : EdtTpEndereco.ItemIndex := 0;
03   'C' : EdtTpEndereco.ItemIndex := 1;
04  end;
```

Nota

Como o valor no banco de dados é representado por apenas um caractere, podemos utilizar o case para esse teste, já que case, no Delphi, funciona apenas para tipos ordinais e char.

Com isso é solucionado o problema da apresentação do ComboBox de tipo de endereço, porém o valor desse combo ainda não está sendo repassado para a query. Portanto vá ao evento *OnChange* do *EdtTpEndereco* e implemente-o como na **Listagem 11**.

Essa implementação verifica qual o *ItemIndex* do *EdtTpEndereco* e atribui o valor correspondente para a coluna *enderecoTP*.

Listagem 11. OnChange do EdtToEndereco

```
01 procedure TfrmPerfil.EdtTpEnderecoChange(Sender: TObject);
02 begin
03   case EdtTpEndereco.ItemIndex of
04    0 : dsEndereco.DataSet.FieldByName('enderecoTP').AsString := 'E';
05    1 : dsEndereco.DataSet.FieldByName('enderecoTP').AsString := 'C';
06   end;
07 end;
```

Como o valor que será armazenado na coluna é o primeiro caractere da palavra, podemos utilizar o código:

```
dsEndereco.DataSet.FieldByName('enderecoTP').AsString:=EdtTpEndereco.Text[1];
```

O código acima é mais limpo, economiza linhas e facilita a leitura, porém só funciona para esse tipo de situação, onde o valor é correspondente a um caractere específico.

Com os problemas solucionados, compile novamente a aplicação e efetue as operações de CRUD desejadas. Observe que, ao tentar deletar, salvar ou cancelar, uma mensagem de confirmação será apresentada, a qual pode ser alterada ou removida na propriedade *Confirmation* do navigator.

Apresentando os produtos dinamicamente

A página de listagem de produtos, apresentada na **Figura 8**, será semelhante às páginas de lojas virtuais famosas. Nela, os produtos devem ser mostrados de forma dinâmica a partir dos registros cadastrados no banco de dados. Para criá-la, adicione um novo frame ao projeto e o renomeie para *frmProdutos*.

Figura 8. Tela da loja

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Após a criação do frame devemos criar uma nova query na *UserSession*, chamada *qryProdutos*, que será importante não só para as consultas dos produtos, mas também para a manipulação do carrinho já que os itens do pedido serão "adicionados" nela.

Para que a *qryProdutos* seja capaz de retornar a lista de produtos e disponibilizar uma relação dos produtos contidos no carrinho de uma forma simples, adicione o código da **Listagem 12** na sua propriedade *SQL*.

Listagem 12. Instrução SQL da qryProdutos

```
01 SELECT
02   PRODUTO.*,
03   0   AS QUANTIDADE,
04   0.00 AS TOTAL
05 FROM PRODUTO
06 WHERE DISPONIVEL
```

Observe que na tabela de produtos existem apenas informações do produto, porém será preciso trabalhar com uma estrutura onde as colunas *quantidade* e *total* estejam disponíveis para armazenar os produtos adicionados pelo cliente. Essa query servirá como uma tabela de memória, e para que isso funcione corretamente é necessário atribuir o valor *false* para a propriedade *CheckUpdatable* para que a query permita gravações em campos de memória. Por fim, para que as alterações feitas a partir dessa query não reflitam no banco de dados, altere a propriedade *CachedUpdates* para *true*.

Em seguida, no *UserSession*, vá ao evento *OnCreate* e adicione o código a seguir para que a *qryProdutos* seja aberta no momento em que o *UserSession* é instanciado, e fique ativa durante toda a aplicação:

```
01 qryProdutos.Close();
02 qryProdutos.Connection := Connection;
03 qryProdutos.Open();
```

Isso é importante porque essa query não tem só as informações dos produtos, ela é a própria lista de compras. Como ela está ativa durante toda a aplicação, fica muito simples apresentar a lista de compras: basta apresentar uma grid ligada a essa query utilizando um filtro em que o campo *quantidade* seja maior que zero.

No *frmProdutos* não será inserido nenhum componente, o que será feito é criar um novo frame que conterá o design dos itens. Então, adicione ao projeto um novo frame e o renomeie para *frmItem*, que será composto por um *TIWImage*, dois *TIWButtons* e cinco *TIWLabels*, resultando em um design próximo ao dos itens apresentados na **Figura 8**. Para o componente de imagem, altere para *false* a propriedade *AutoSize*, e no código adicione as seguintes propriedades:

```
property idProduto : integer read fIdProduto write fIdProduto;
property dataSet : TDataSet read fDataSet write fDataSet;
```

Essas propriedades servirão para que os botões de adicionar e remover um item do carrinho saibam qual dataset será manipulado e qual é o item que esse formulário está representando. Para adicionar ou remover um item será utilizado o método *SetQuantidade*, que deve ser adicionado na seção *private* e implementado como a **Listagem 13**.

Listagem 13. Método responsável por adicionar ou remover um item ao pedido

```
01 procedure TfrmItem.SetQuantidade(pnQtd: Integer);
02 Var qtdAtual : Integer;
03 begin
04   dataSet.Locate('produtoId',IdProduto,[]);
05
06   qtdAtual := dataSet.FieldByName('quantidade').AsInteger;
07
08   if ((pnQtd = -1) and (qtdAtual = 0) ) then
09    exit;
10
11   dataSet.Edit;
12   dataSet.FieldByName('quantidade').AsInteger := qtdAtual + pnQtd;
13   dataSet.FieldByName('total').AsFloat := dataSet.FieldByName('quantidade').
     AsInteger * dataSet.FieldByName('preco').asFloat;
14   dataSet.Post;
15
16   lblQtd.Caption := dataSet.FieldByName('quantidade').AsString;
17 end;
```

O código dessa listagem é bem simples: na linha 4 o dataset é posicionado no item desejado através do método *Locate*, que recebe o id do produto passado por parâmetro. Na linha 8, a fim de evitar quantidades negativas no carrinho, é verificado se a quantidade do item no carrinho é igual a zero e se o parâmetro *pnQtd* é igual a "-1", o que indica a subtração de um item. Em seguida, se o *if* da linha 8 não for satisfeito, é feito um *Edit* no dataset alterando o valor da quantidade, do total e atualizando o label que apresenta o número de itens.

Com o frame de item pronto, devemos criar dinamicamente um item para cada registro da tabela de produto. Para isso, retorne ao *frmProdutos* e adicione as seguintes constantes:

```
MARGIN_TOP   = 50;
ARGIN_LEFT  = 20;
COLUNAS     = 4;
MARGIN_ENTRE_ITENS = 10;
```

Essas constantes servem para facilitar a manutenção, já que a partir delas é que serão determinadas as margens das bordas, o número de colunas e a margem entre um item e outro. Em seguida, crie o método *ExibirItens* (veja a **Listagem 14**).

Esse método faz um loop sobre os produtos e, para cada registro, instancia um *TfrmItem* e posiciona-o na página de acordo com as constantes. As variáveis *nTop* e *nLeft*, inicializadas com os valores definidos nas constantes *MARGIN_TOP* e *MARGIN_LEFT*, servem para controlar o posicionamento de cada item e, por isso, devem ser devidamente incrementadas. A variável *nLeft* é incrementada a cada loop com a largura do componente e a margem

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

configurada. No entanto, caso ela atinja o número de colunas desejadas (linha 25), será reiniciada com o valor da margem da página e o *nTop* será incrementado.

Listagem 14. Método responsável por carregar os produtos dinamicamente

```
01 procedure TfrmProdutos.ExibirItens;
02 Var
03  Item  : TfrmItem;
04  nTop  : Integer;
05  nLeft : Integer;
06 begin
07  nTop  := MARGIN_TOP;
08  nLeft := MARGIN_LEFT;
09
10  qryProdutos.First;
11  while Not(qryProdutos.Eof) do
12  Begin
13   Item  := TfrmItem.Create(self);
14   Item.Name  := 'Item_' + qryProdutos.RecNo.ToString();
15   Item.Parent := IWFrameRegion;
16   Item.IWFrameRegion.Top   := nTop;
17   Item.IWFrameRegion.Left  := nLeft;
18
19   Item.dataSet  := qryProdutos;
20   Item.IdProduto := qryProdutos.FieldByName('produtoId').AsInteger;
21
22   inc(nLeft,Item.Width + MARGIN_ENTRE_ITENS);
23   qryProdutos.Next;
24
25   if ((qryProdutos.RecNo-1) mod COLUNAS = 0) then
26   Begin
27    inc(nTop,Item.Height + MARGIN_ENTRE_ITENS);
28    nLeft := MARGIN_LEFT;
29    continue;
30   End;
31  End;
32 end;
```

Na linha 13 a variável *item* recebe uma instância de *TfrmItem*, e na linha 15 o seu *parent* recebe a region do frame de produtos. Por fim, nas linhas 19 e 20 são atribuídos o dataset e o id do produto que a instância do item irá controlar.

Ainda resta um detalhe para que a tela de produtos seja concluída: falta preencher os campos do item. Para isso, retorne ao *frmItem* e implemente o método *PreencherDados* como a **Listagem 15**.

Na linha 5 é feito o posicionamento sobre o item com o uso do *Locate*. Note que nesse momento é necessário ter o dataset e *IdProduto* preenchidos. Da linha 6 à 8 os valores dos fields do dataset são atribuídos aos capitions. Para a atribuição da imagem será necessário um pouco mais de esforço, por isso se utiliza a variável *imgTemp*, que recebe a imagem do banco de dados através do *typecast TBlobField*. Em seguida, o conteúdo dessa variável é passado para a *image* através do *Assign*, e, ao fim, a referência de *imgTemp* é liberada da memória.

Para concluir essa etapa, basta fazer a chamada do método *PreencherDados*. Ela será realizada no método *set* da propriedade *IdProduto*, garantindo que sempre que um novo id for atribuído ao componente, os dados sejam atualizados. Crie esse método conforme a **Listagem 16**, e note que ele não faz só a chamada ao *PreencherDados*, mas também valida se o dataset já foi atribuído, evitando uma possível violação de acesso.

Listagem 15. Método para preencher os campos do *frmItem*

```
01 procedure TfrmItem.PreencherDados;
02 var
03  imgTemp : TJPegImage;
04 begin
05  dataSet.Locate('produtoId',IdProduto,[]);
06  lblNomeProduto.Caption  := dataSet.FieldByName('Nome').AsString;
07  lblPreco.Caption        := dataSet.FieldByName('preco').AsString;
08  lblQtd.Caption          := dataSet.FieldByName('quantidade').AsString;
09
10  imgTemp := TJPEGImage.Create;
11  try
12   imgTemp.Assign(TblobField(dataSet.FieldByName('imagem')));
13   image.Picture.Bitmap.Assign(imgTemp);
14  finally
15   FreeAndNil(imgTemp);
16  end;
17 end;
```

Listagem 16. Método set da propriedade IdProduto

```
01 procedure TfrmItem.SetIdProduto(const Value: integer);
02 begin
03  fIdProduto := Value;
04  if Assigned(dataSet) then
05   PreencherDados();
06 end;
```

Com a página de produtos pronta, retorne ao menu e faça a chamada ao método *AbrirFrame*, passando o *TfrmProdutos* (veja a **Listagem 17**). Depois, chame o método *consultarProdutos*, que deve ser implementado como a **Listagem 18** e é responsável por desabilitar os filtros da query de produtos e chamar o *exibirItens*.

Listagem 17. Chamada da tela de produtos

```
01 procedure TfrmMenu.Produtos1Click(Sender: TObject);
02 var frmProdutos : TfrmProdutos;
03 begin
04  frmProdutos := TfrmProdutos(AbrirFrame(TfrmProdutos));
05  frmProdutos.consultarProdutos;
06 end;
```

Listagem 18. Método ConsultarProdutos

```
01 procedure TfrmProdutos.ConsultarProdutos;
02 begin
03  qryProdutos := UserSession.qryProdutos;
04  qryProdutos.Filtered := false;
05  exibirItens();
06 end;
```

Tela de carrinho de compras

A venda será finalizada a partir do menu do carrinho, no qual o cliente poderá ver o que escolheu comprar e o total da compra. Então, adicione um novo frame, o renomeie para *frmCarrinho* e edite-o para que fique semelhante à **Figura 9**.

No componente grid, defina os campos a serem apresentados na lista de produtos comprados, e, em seguida, vá ao evento *OnCreate* do *IWFrameRegion* e implemente-o da seguinte forma:

```
dsProduto.DataSet := UserSession.qryProdutos;
UserSession.FilterItensNoCarrinho();
lblTotal.Caption := FloatToStr(GetTotalCarrinho());
```


DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

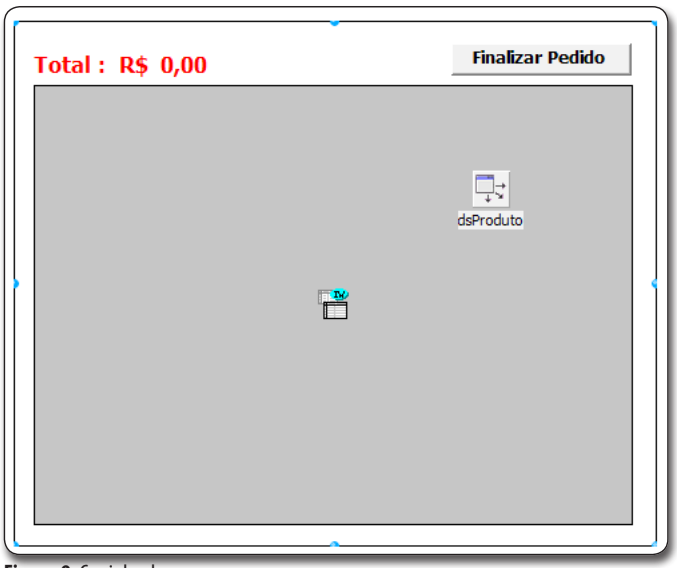

Figura 9. Carrinho de compras

Ao instanciar o *frmCarrinho*, será feita a referência da query de produtos ao dataset que irá controlar a grid, e em seguida será chamando o método *FilterItensNoCarrinho* da *UserSession* (veja a **Listagem 19**). Esse método será responsável por filtrar apenas os itens que foram adicionados ao carrinho, ou seja, os itens com *quantidade* maior que zero. No final do código anterior, o valor resultante do método *GetTotalCarrinho*, responsável pela atualização do valor total, é atribuído ao caption do label *lblTotal*. Como esse método ainda não foi criado, implemente-o como mostrado na **Listagem 20**.

Para pegar o total, o dataset é percorrido e o valor da coluna *Total* é incrementado à variável *Result*. Note que como o dataset está filtrado, não será necessário verificar se a quantidade está ou não preenchida.

Listagem 19. Método para filtrar os itens no carrinho

```
01 procedure TIWUserSession.FilterItensNoCarrinho;
02 begin
03   qryProdutos.Filtered := false;
04   qryProdutos.Filter := 'quantidade > 0';
05   qryProdutos.Filtered := true;
06 end;
```

Listagem 20. Método para pegar o total da compra

```
01 function TfrmCarrinho.GetTotalCarrinho: Real;
02 begin
03   dsProduto.DataSet.First;
04   Result := 0.00;
05   while Not(dsProduto.DataSet.Eof) do
06   begin
07     Result := Result + dsProduto.DataSet.FieldByName('Total').AsFloat;
08
09     dsProduto.DataSet.Next;
10   end;
11 end;
```

Finalizando a venda

Com a tela de carrinho concluída, adicione um novo frame ao projeto e nomeie-o como *TfrmFinalizarVenda*. Ele será chamado no evento *OnClick* do botão de finalização de venda da tela de carrinho e fará a chamada ao método *AbrirFrame* do *frmMenu*, passando o *TfrmFinalizarVenda* como parâmetro conforme o código a seguir:

```
TfrmMenu(Owner).AbrirFrame(TfrmFinalizarVenda);
```

Agora, modifique o frame recém-criado para que o mesmo tenha um aspecto semelhante à **Figura 10**, na qual estarão presentes a opção de forma de pagamento e a lista de endereços para que o cliente selecione o endereço desejado para entrega.

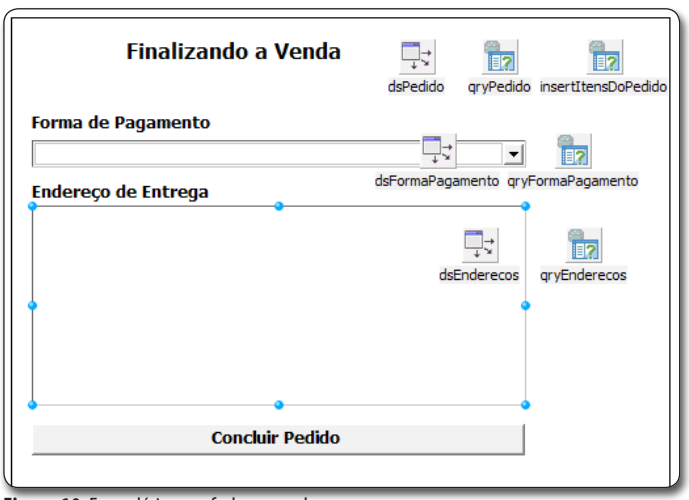

Figura 10. Formulário para fechar a venda

Essa página conterá quatro componentes *TFDQuery*: o *qryPedidos* e o *qryFormaPagamento* são os mais simples, e suas propriedades SQL possuem as instruções *SELECT * FROM COMPRA* e *SELECT * FROM FORMAPAGAMENTO*, respectivamente. Já o *qryEnderecos* e o *insertItensDoPedido* são responsáveis, respectivamente, por retornar os endereços de entrega cadastrados e inserir os itens de um pedido na tabela *compraitem*. As **Listagens 21** e **22** mostram os comandos SQL que devem ser inseridos nesses dois últimos componentes TFDQuery.

Depois, faça a ligação das queries com os datasets e, na sequência, dos campos com os datasets e fields. A forma de pagamento é um *TIWDBLookupComboBox* e a lista de endereços, um *TIWDBLookupListBox*. Esses componentes devem ser ligados ao DataField e ao DataSource correspondente, não se esquecendo das propriedades *ListSource* e *ListField*.

No *OnCreate* do region, abra as queries e ligue-as com o *Connection* de *UserSession*. Na sequência, codifique o método *FinalizarVenda* como na **Listagem 23** e chame-o no *OnClick* do botão concluir pedido.

O método *FinalizarVenda* é bem trivial. Primeiro, o id do cliente e a data são passados para a *qryPedido*, e em seguida é feito um post nessa query.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 21. Instrução SQL para retornar endereços cadastrados como de entrega

```
01 SELECT
02  ENDERECOID,
03  CONCAT(CEP,' - ',
04        LOGRADOURO,',',
05        BAIRRO,' Nº ',NUMERO) AS ENDERECO
06 FROM ENDERECO
07 WHERE CLIENTEID =:CLIENTEID
08 AND ENDERECOTP = 'E'
```

Listagem 22. Instrução SQL para inserir os itens de um pedido

```
01 INSERT INTO compraitem(
02  compraid ,
03  produtoid ,
04  quantidade,
05  total
06 )VALUES(
07  :compraid ,
08  :produtoid ,
09  :quantidade,
10  :total
11 )
```

Listagem 23. Método resposável pela finalização da venda

```
01 procedure TfrmFinalizarVenda.FinalizarVenda;
02 begin
03  qryPedido.FieldByName('ClienteId').AsInteger := UserSession.idUsuario;
04  qryPedido.FieldByName('data').AsDateTime := now();
05  qryPedido.Post;
06
07  UserSession.FilterItensNoCarrinho;
08  UserSession.qryProdutos.First;
09  While Not(UserSession.qryProdutos.Eof) do
10  begin
11   insertItensDoPedido.Close;
12   insertItensDoPedido.paramByName('comprald').AsInteger :=
     qryPedido.FieldByName('comprald').AsInteger;
13   insertItensDoPedido.paramByName('produtoId').AsInteger :=
     UserSession.qryProdutos.FieldByName('produtoId').AsInteger;
14   insertItensDoPedido.paramByName('quantidade').AsInteger :=
     UserSession.qryProdutos.FieldByName('quantidade').AsInteger;
15   insertItensDoPedido.paramByName('total').AsFloat  :=
     UserSession.qryProdutos.FieldByName('total').AsFloat;
16   insertItensDoPedido.ExecSQL;
17
18   UserSession.qryProdutos.Next;
19  end;
20
21  UserSession.qryProdutos.Filtered := false;
22  UserSession.ClearVenda;
23
24  TfrmMenu(Owner).AbreFrame(TfrmHome);
25 End;
```

Com a gravação do pedido feita, um loop percorrendo todos os itens do carrinho é realizado para começarmos a incluí-los no banco. Para cada pedido, seus valores, obtidos da *qryProdutos* da *UserSession* e da *qryPedido*, são passados para os parâmetros da query, que é executada no final de cada laço do loop através do método *ExecSQL*. Note que o campo *compraId* obtido da *qryPedido* é o id do pedido recém-adicionado ao banco na linha 5.

No final é feita a chamada ao método *ClearVenda* da *UserSession*, que somente fará um *close* e um *open* na tabela de produtos. Como essa tabela não está sendo persistida, no processo de reabertura seus dados são perdidos. O *TfrmHome* chamado no final da venda pode ser qualquer frame desejado, como uma tela de agradecimento da compra ou a página principal da aplicação. Com isso, finalizamos a codificação do projeto.

Templates HTML

O IntraWeb dispõe de um recurso muito interessante, que é a opção de se criar templates em HTML e carregar os componentes instanciados no formulário para o template. Entre as vantagens do seu uso podemos destacar:

1. O programador não precisa se preocupar com a montagem e organização do formulário, deixando essa tarefa para a equipe de design, que lida com HTML e CSS para criar os templates;
2. É possível melhorar o visual da aplicação e a organização dos elementos;
3. O uso de componentes visuais é reduzido, ou seja, elementos que não terão interação com o código podem ser descartados do dfm, como imagens estáticas, labels descritivos e regions utilizados só para alinhamentos.

A seguir criaremos um template para a tela de login do nosso projeto, que se chama *frmLogin*. Os templates devem ser salvos dentro da pasta *templates* no diretório da aplicação, com o mesmo nome do formulário e com a extensão ".html". Portanto, crie um arquivo HTML chamado *frmLogin.html* dentro da pasta *template*, no output do projeto. No corpo do HTML desse arquivo deve ser implementado o código da **Listagem 24**.

Como pode ser observado, o código é um HTML padrão, porém, onde se deseja renderizar um componente do IntraWeb, utiliza-se a tag {%<NomeComponente>%}.

Listagem 24. Instrução HTML para o template da tela de Login

```
01 <body style="background: #373737">
02   <div class="centralPanel">
03     <div>
04       <h1 class="title">Login ...</h1>
05     </div>
06
07     <div class="login">
08       <p>
09         <label>Email:</label>
10         <span style="padding-right: 20px">{%edtUsuario%}<span>
11       </p>
12
13       <p>
14         <label>Password:</label>
15         {%edtUsuario%}
16       </p>
17
18       <p class="login-submit">
19       {%btnEntrar%}
20       </p>
21     </div>
22   </div>
23 </body>
```


DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Para que esse HTML seja utilizado como template, no formulário desejado devemos adicionar o componente *TIWTemplateProcessorHTML* e na proriedade *LayoutMgr* do formulário deve ser feita a referência ao *TemplateProcessor*. Se desejar melhorar o visual, adicione um código CSS à página HTML, por exemplo o apresentado na **Listagem 25**.

Listagem 25. CSS do template de login

```
<style>
  .centralPanel{
    padding: 0px 20px 20px 20px;
    width:280px;
    height:280px;
    position:absolute;
    top:20%;
    left:40%;
    background: #2F4F4F;
    border-radius:25px;
  }
  .title{
    height:40px;
    background: #696969;
    padding-left: 35%;
  }
</style>
```

Para finalizar a aplicação com o template, basta compilar o projeto. O resultado obtido deve ser semelhante ao da **Figura 11**, que é bem satisfatório levando em conta a quantidade de código HTML e CSS utilizado. Porém, o visual da aplicação ainda pode ser melhorado, e é isso que veremos no próximo tópico.

Figura 11. Formulário de login com template

Melhorando o visual

O IntraWeb disponibiliza algumas maneiras para melhorar o visual da aplicação. Uma delas foi vista na seção anterior, o template, mas outras também podem ser utilizadas, como frameworks CSS como o Bootstrap e Materialize.

CSS

Para exemplificarmos melhor o uso de CSS, crie um arquivo chamado *default.css* na pasta *wwwroot/Files/css*, dentro da pasta de output do projeto, e implemente nesse arquivo o seguinte código:

```
.CONTAINERCSS{
    border-radius: 25px;
    background-color: #696969;
}
```

Sempre que desejar alterar um componente específico, crie instruções CSS para o elemento ou para sua classe. A classe de um componente pode ser definida através de sua propriedade *css*, disponível na maioria dos componentes do IntraWeb.

Pode-se adicionar um arquivo CSS à aplicação de três formas. A primeira é referenciar o arquivo no formulário que deseja carregá-lo, utilizando para isso a propriedade *StyleSheet* do formulário, na qual deve ser informada a URL do arquivo *.css* ou o path do mesmo. Para informar o path, utilize a opção *Filename*, na qual deve ser informada apenas a estrutura abaixo da pasta *Files*, que no nosso exemplo é: *css/default.css*. Ao compilar o projeto com essa opção configurada, deve-se obter um formulário de login com as bordas arredondadas.

Outra maneira de adicionar uma folha de estilo é através da propriedade *StyleSheet* da unit *ServerController*. A diferença entre adicionar na *ServerController* ou em um formulário, como feito anteriormente, é que na *ServerController* temos nosso CSS disponível em nível global, sendo carregado, portanto, para todos os formulários.

Como pode ser observado, é possível adicionar apenas um arquivo para essa propriedade. Mas se o CSS estiver separado em vários arquivos, como fazer? A terceira forma resolve esse problema: na unit *ServerController* existe uma propriedade chamada *ContentFiles*, na qual podem ser adicionados vários arquivos por meio de uma lista de string. Exemplo: *Files/css/default.css*.

Bootstrap

Existem várias vantagens em utilizar o Bootstrap, entre as quais destacam-se a responsividade e um belo design. Além disso, o framework oferece vários componentes prontos, o que facilita muito na hora do desenvolvimento. No entanto, nem todos os recursos do Bootstrap podem ser utilizados corretamente no projeto, já que muitas vezes o framework exige um HTML com estrutura diferente da gerada pelo IntraWeb, a qual não pode ser modificada. O *dropdown* é um bom exemplo desse caso, mas outros componentes, como edits, labels, botões e componentes de estrutura mais simples ou que dependem apenas da referência de uma classe, funcionam perfeitamente.

Para adicionar o Bootstrap ao projeto, basta obter seu pacote na versão desejada no site oficial (seção **Links**) e adicioná-lo na pasta *css*, em *wwwroot/Files*. Em seguida, deve-se importar os arquivos do pacote para o projeto, utilizando uma das opções já explicadas para a carga de CSS. A mais indicada é através da propriedade *ContentFiles*, já que ela também permite adicionar arquivos JS.

Esses arquivos JS podem ser arquivos JavaScript criados pelo desenvolvedor, arquivos do próprio framework, ou ainda do jQuery, que é uma biblioteca utilizada pelo Bootstrap. Esses dois últimos tipos de arquivos são necessários para muitos componentes do

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Bootstrap, que, sem eles, serão apresentados de maneira incorreta ou não serão apresentados. Neste pequeno exemplo não nos preocupamos com arquivos JS, já que os componentes utilizados funcionam bem sem eles.

Então, com o Bootstrap no projeto, bastar alterar as propriedades *CSS* para a classe desejada. Como exemplo será utilizada novamente a tela de login, portanto referencie no formulário o arquivo *bootstrap.css*, e nos labels da tela altere a propriedade CSS para "label label-default", os edits para "form-control" e o botão para "btn btn-primary". Esses valores dizem respeito às classes que serão adicionadas a um componente quando ele for renderizado, como já explicado. A diferença aqui é que essas classes são utilizadas pelo framework para adicionar estilos e comportamentos específicos aos componentes. Elas não serão abordadas neste artigo pois não estão dentro de seu escopo, mas para saber mais sobre elas acesse a documentação do Bootstrap disponível no seu site oficial (seção **Links**).

Ao executar o projeto, observe que os componentes tiveram seu design alterado, resultando no visual da **Figura 12**.

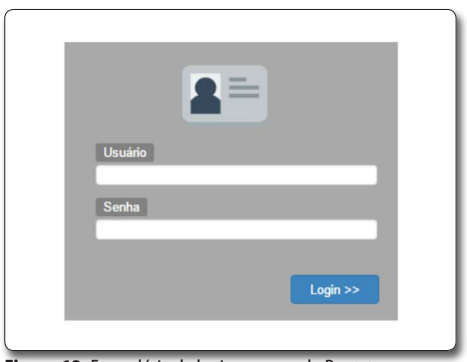

Figura 12. Formulário de login com uso do Bootstrap

Materialize

O Materialize, assim como o Bootstrap, é um framework CSS que oferece o recurso da responsividade e melhorias visuais, e pode ser integrado ao projeto IntraWeb com a injeção de seus arquivos CSS na pasta *wwwroot/Files/css* do output do projeto. Ele pode ser obtido no seu site oficial (veja a seção **Links**), e sua forma de utilização será basicamente a mesma que a do Bootstrap, porém os nomes das suas classes serão diferentes.

O Materialize tem uma estrutura mais flexível que a do Bootstrap, o que lhe garante uma maior compatibilidade com os componentes do IntraWeb. Para testar isso, referencie no formulário de login o arquivo *materialize.css*, altere a propriedade *CSS* dos edits para "validate" e a do botão para "btn". Compile o projeto e observe que o design ficou semelhante ao da **Figura 13**.

Se desejar obter melhores resultados, trabalhe de forma mista: use templates com CSS e mais um dos frameworks citados anteriormente (Bootstrap ou Materialize).

Além do uso de CSS, é possível adicionar componentes, como o IWBootstrap e CGDevtools, que disponibilizam paletas específicas para o desenvolvimento, cujos elementos já contam com implementações de responsividade e melhoria visual.

Figura 13. Formulário de login com uso do Materialize

O IWBootstrap disponibiliza componentes já implementados com o Bootstrap de forma nativa, e com isso elementos que teriam problemas para funcionar com Bootstrap se tornam mais simples. Já o CGDevtools disponibiliza elementos com o uso de jQuery, que em sua maioria têm efeitos visuais, como transparência, fade in e fade out. Para profissionais sem conhecimento em HTML, CSS ou em um dos frameworks citados, é altamente recomendado que se faça uso de um desses pacotes.

O desenvolvimento com IntraWeb ainda vai muito além do apresentado neste artigo. É possível criar eventos com JavaScript, renderizar HTML em tempo de execução, entre outros recursos. Tente, a partir do conhecimento passado com este artigo, estudar e implementar o que aqui foi visto e ir sempre além do básico.

Autor

Gutierry Antonio
gutierrydsn@hotmail.com

Entusiasta de Big Data e Data Warehouses, atua como Engenheiro de Software e DBA MySQL. É graduado em Sistemas de Informação e Análise de Sistemas, e atualmente cursa especialização em Engenharia de Sistemas. Possui experiência em desenvolvimento com Delphi, Ruby on Rails e com bancos de dados Firebird e MySQL, além de possuir conhecimento em Java, C#, C++ e nos bancos de dados SQL Server e Oracle, bem como em AWS.

Links:

IWBootstrap Framework – Demo do IWBootstrap Framework
http://www.atozed.com/index.EN.aspx

Atozed – Download do IntraWeb
http://www.atozed.com/IntraWeb/Download/index.EN.aspx

CGDevtools
http://www.cgdevtools.com/

TMS Software – Componentes TMS para IntraWeb
http://tmssoftware.com/site/products.asp?t=iw

Site official do Bootstrap
http://getbootstrap.com/

Site oficial do Materialize
http://materializecss.com/

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,
assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486